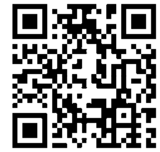
软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(12):4688−4703 [doi: 10.13328/j.cnki.jos.006354]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

ALERT: 基于 Radix Tree 的工作负载自适应学习型索引*



陈井爽[1,2], 陈 珂[1,2], 寿黎但[1,2], 江大伟[1,2], 陈 刚[1,2]

[1](浙江大学 计算机科学与技术学院, 浙江 杭州 310027)

[2](浙江省大数据智能计算重点实验室(浙江大学), 浙江 杭州 310027)

通信作者: 寿黎但, E-mail: should@zju.edu.cn

摘 要: 学习型索引通过学习数据分布可以准确地预测数据存取的位置, 在保持高效稳定的查询下, 显著降低索引的内存占用. 现有的学习型索引主要针对只读查询进行优化, 而对插入和更新支持不足. 针对上述挑战, 设计了一种基于 Radix Tree 的工作负载自适应学习型索引 ALERT. ALERT 使用 Radix Tree 来管理不定长的分段, 段内采用具有最大误差界的线性插值模型进行预测. 同时, ALERT 使用一种高效的插入缓冲来降低数据插入更新的代价. 针对点查询和范围查询提出两种自适应重组优化方法, 通过对工作负载进行感知, 动态地调整插入缓冲的组织结构. 经实验验证, ALERT 与业界流行的学习型索引相比, 构建时间平均降低了 81%, 内存占用平均降低了 75%, 在保持了优秀读性能的同时, 使插入延迟平均降低了 50%; 此外, ALERT 使用自适应重组优化能有效感知查询工作负载特征, 与不使用自适应重组优化相比, 查询延迟平均降低了 15%.

关键词: 学习型索引; 自适应索引; 机器学习; 数据库
中图法分类号: TP311

中文引用格式: 陈井爽, 陈珂, 寿黎但, 江大伟, 陈刚. ALERT: 基于 Radix Tree 的工作负载自适应学习型索引. 软件学报, 2022, 33(12): 4688–4703. http://www.jos.org.cn/1000-9825/6354.htm

英文引用格式: Chen JS, Chen K, Shou LD, Jiang DW, Chen G. ALERT: Workload-adaptive Learned Index Based on Radix Tree. Ruan Jian Xue Bao/Journal of Software, 2022, 33(12): 4688–4703 (in Chinese). http://www.jos.org.cn/1000-9825/6354.htm

ALERT: Workload-adaptive Learned Index Based on Radix Tree

CHEN Jing-Shuang[1,2], CHEN Ke[1,2], SHOU Li-Dan[1,2], JIANG Da-Wei[1,2], CHEN Gang[1,2]

[1](College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China)

[2](Key Laboratory of Big Data Intelligent Computing of Zhejiang Province (Zhejiang University), Hangzhou 310027, China)

Abstract: Learned indexes are capable of predicting the accurate location of data in storage by learning the data distribution. These indexes can significantly reduce storage consumption while providing efficient query processing. Existing learned indexes are mostly optimized for read-only queries, but inadequate in supporting insertions and updates. In an attempt to address the challenges faced by learned index, this study proposes a workload-adaptive learned index named ALERT. Generally, ALERT employs a Radix Tree to manage variable-length segments, where each segment contains a linear interpolation model with a maximum error-bound. Meanwhile, ALERT utilizes an insertion memory buffer to reduce the cost of updates. Following the database-cracking approach, the study proposes adaptive index maintenance during the run-time processing of point queries and range queries. The maintenance technique is implemented by performing workload-aware dynamic re-organization on the insertion buffer. Experimental results confirm that, when compared to state-of-the-art learned index, ALERT achieves competitive results as it reduces the index's average construction time by 81%, the average memory utilization by 75%, the average latency of insert by 50%, while maintaining competitive read performances. The average query latency of ALERT is also reduced by 15%, owing to its effective workload-aware optimization.

Key words: learned index; adaptive index; machine learning; database

* 基金项目: 浙江省重点研发计划(2021C01009); 国家自然科学基金(62050099); 浙江省自然科学基金(LY18F020005)
收稿时间: 2021-01-26; 修改时间: 2021-04-12; 采用时间: 2021-04-26; jos 在线出版时间: 2022-05-24

索引是实现高效数据访问最重要的技术, 被广泛地用于数据库系统. 现实生活中, 数据库的应用场景各式各样, 不同的应用场景具有不同的数据分布. 传统的索引方法往往针对于某种特定场景进行优化或者对于各种场景都进行了广泛地权衡, 而没有考虑数据的实际分布. 如果知道了数据的确切分布, 现有的索引方法几乎都还可以做更进一步的优化. 2018 年, Kraska 等人[1]提出了学习型索引的概念, 探索使用机器学习模型来模拟数据分布, 证明了学习型索引可以通过利用数据分布降低索引内存占用, 并且提高索引在特定分布数据集上的查询性能, 从而为高效的索引设计提供了一种新的研究方向.

但学习型索引最大的问题在于对插入和更新的支持较弱, 这大大限制了学习型索引的使用场景. 学习型索引的插入和更新一直都是一项具有挑战性的工作[2,3], 为了支持插入和更新, 有两种常用的方法: 一种是原地更新(in-place)[4,5]的方式, 给键值数组分配冗余的空间, 当新的数据插入时, 通过键值移动来容纳插入; 另一种是使用单独缓冲区(delta buffer)[4,6,7]的方式, 使用一个单独的缓冲区缓冲新的插入, 当缓冲区中的键值达到一定数量时, 再与底层数组合并. 学习型索引通常使用一个较大的排序键值数组存储底层数据, In-place 的方式会带来大量的键值移动, Delta Buffer 则需要容量较大的缓冲区来均摊合并的代价. 无论是大量的键值移动还是容量较大的缓冲区, 都会带来较大开销, 增加各项操作的延迟. 因此, 降低数据插入更新的成本, 对于学习型索引在数据库中的应用十分迫切.

为了减少索引的内存占用, 同时保持高效稳定的查询性能, 并且降低插入更新的成本, 本文设计了一种基于 Radix Tree[8]的工作负载自适应学习型索引 ALERT(a workload-adaptive learned index based on radix tree). ALERT 具有两层结构.

- 底层使用一种基于贪心策略的分段算法, 利用数据分布将数据划分为多个不定长分段, 尽可能地减少分段的数量; 段内采用具有最大误差界的线性插值模型学习数据的分布, 用于高效地预测, 保证稳定且有界的段内搜索性能;
- 上层使用 Radix Tree 来管理底层分段, 因为底层分段的数量相对于原始数据量大大减少, 在数据量较小的情况下, Radix Tree 能够较好地支持查询插入等操作. 此外, Radix Tree 在键长度较短时十分高效, 与学习型索引所适用的数值类型键长较短的特点相契合.

ALERT 使用一种插入缓冲的方式, 避免了插入时的数据移动, 并且尽可能地减少缓冲区中键值数目过多对各项操作的影响. 随着数据的大量插入, 索引的查询性能会受到影响, 因此需要对新插入的数据进行重组优化. 然而, 在缺乏查询工作负载信息的情况下进行数据重组工作具有盲目性. 在现实场景中, 查询工作负载往往具有一定的偏态分布, 某些数据被访问的频率要高于其他数据, 而某些数据很少会被查询涉及. 如果后续的查询不会访问到重组涉及的数据, 那么会带来大量不必要的开销. 因此, ALERT 使用两种自适应重组优化方法, 有针对性地对工作负载中点查询和范围查询涉及的热点数据进行重组优化.

本文实现了 ALERT 的原型, 并进行了大量的实验和比较. 实验表明: ALERT 具有较高的构建效率, 也保证了相对高效的点查询和范围查询性能, 并且能够很好地支持插入和更新, 与业界流行的学习型索引相比, 构建时间平均降低了 81%, 内存占用平均降低了 75%, 在保持了优秀的读性能的同时, 使得插入延迟平均降低了 50%; 同时, 通过感知查询工作负载有针对性地进行重组优化, 以较低的代价降低了插入对于索引性能的影响, 改善了索引的性能, 相对于不使用自适应重组优化, 使查询延迟平均降低了 15%.

1 相关工作

1.1 传统索引

B 树[9]和 B+树[10]被设计用于加速数据库系统上的搜索. 为了降低存储成本,提高访问效率, 人们提出了许多变种. B+树在现代处理器上具有良好的缓存行为: 如果将 B+树节点的大小与缓存线对齐, 那么节点中的所有数据都会在一条缓存线中被访问. 为了利用这个特性, Rao 等人提出了可以高效利用缓存的 B+树变体: CSS-tree[11]和 CSB+-tree[12], 从而提供更快的查询性能. 另一方面, 随着硬件的发展, 现代处理器将多个核心集成于一个芯片中, 并且每个核都配有矢量(SIMD)单元, 另外, GPU, FPGA 等异构加速硬件的性能也越来越

© 中国科学院软件研究所 http://www.jos.org.cn

高. 为了利用现代处理器不断增长的计算能力, FAST[13]根据架构特性组织排布元素和树的层级结构, 从而利用 SIMD 指令和 GPU 来加速查询的性能.

Radix Tree 是数据库中另一种常用的索引结构, 可以看作是一种以二进制位为关键字的特殊前缀树. Radix Tree 有许多有趣且有用的特性: 查询插入性能只取决于键的长度, 而不取决于数据量; 键值按顺序存储, 从而支持范围查询. 然而, 对于特定数据分布来说, Radix Tree 的内部节点会具有许多稀疏分布的分支, 从而造成过多的内存消耗和糟糕的缓存利用率. ART (the adaptive radix tree)[14]解决了 Radix Tree 内部节点稀疏的问题. ART 的节点的大小与节点中键空间填充的密度成比例, 使用自适应大小的节点来减少稀疏节点造成的内存浪费. 一棵 ART 可以有 4 种不同的节点类型, 并根据非空子节点的数量选择节点类型. 为了优化稀疏的区域, ART 使用了路径压缩, 通过压缩只有一个子节点的所有内部节点, 可以显著降低树的高度和空间消耗. 因此, ART 在数据库系统中具有广泛的应用, 是数据库系统的首选索引结构之一, 包括一些完全成熟的数据库管理系统, 如 HyPer[15]和 DuckDB[16], 也使用了 ART.

1.2 学习型索引

学习型索引背后的思想在于: 索引可以看作从查找键到存储位置的近似函数, 从而将索引的构建问题转换为使用机器学习模型来学习数据累积分布的问题. Kraska 等人[1]提出的学习型索引 RMI (recursive model index)将数据递归地进行分区, 在一个分区中使用模型学习数据的一个子集, 从而建立一个按阶段组织的具有层次结构的模型, 在每个阶段使用模型的预测结果选择下一阶段的模型, 直到最后一个阶段预测查找键在排序数组中的位置. 为了保证索引能够准确地找到所查询的键值, 需要记录下最后一个阶段中的每个模型对于所有数据的预测位置与其真实位置的最小误差和最大误差, 并使用二分搜索进行修正. RMI 利用数据分布加快查询的速度, 降低索引的大小, 但仍存在较多问题: 首先, 构建较为耗时, 并且需要复杂的调优; 其次, 采用自顶向下的构建算法, 盲目地将数据分配给下面的模型, 从而造成了模型的冗余或过载, 并且不具有最差情况下的性能保证; 最后, RMI 是静态索引, 缺乏对动态插入更新的支持. RS (radix spline)[17]是另一种静态学习型索引. RS 提出的目的是为了更快地构建学习型索引, 并且保持优秀的索引大小和查询性能. 与 RMI 不同, RS 自底向上构建索引, 这避免了自顶向下构建时给底层模型分配数据的盲目性, 实现了只需遍历一次数据的高效构建. 但 RS 也有一些缺点, 容易受到数据分布偏移的影响, 当具有离群点时, RS 使用的 Radix Table 会变得很大, 从而大大增加了索引的内存占用.

RMI 等静态学习型索引提出后, 后续的研究主要集中在解决学习型索引对插入和更新的动态支持上. 现有动态学习型索引的解决方案大致有两种: 一种是给底层数组的存储节点分配冗余的空间, 用于容纳新的插入, 如 ALEX[6]为叶子节点分配冗余的空间, 将叶子节点配置为 Gapped Array[18], 当新键插入时, 插入的键会直接放到模型预测的位置, 如果该位置被占用, 则通过移动为新键的存入腾出空间; 另一种方法是使用传统索引作为 Delta Buffer, 对学习型索引中的新插入的键值进行缓冲, 然后批量合并至学习型索引, 以均摊单次插入的开销, 如 Fitting-Tree[4], ASLM[6]和 Dabble[7]等. PGM-index[19]采用的方法有所不同, 静态版本的 PGM-index 在理论上具有最差情况下的性能保证, 动态版本则基于静态的 PGM-index 构建. PGM-index 应用一种指数级方法[20,21], 定义一系列静态 PGM-index, 分别建立在大小为 $2^0,2^1,\ldots,2^b$ 的集合 $S_0,S_1,\ldots,S_b$ 上, 其中, $b=\Theta(\log n)$. 每插入一个键 key, 查找第 1 个空的集合 S_i, 并在集合 $S_0\cap S_1\cap\ldots\cap S_b\cap\{k\}$上构建一个新的 PGMindex, 合并后的新集合包含 2^i 个键, 作为新的 S_i, S_i之前的 i 个集合被清空. 但动态 PGM-index 相对于静态版本牺牲了查询性能, 因为需要搜索 b 个静态的 PGM-index.

2 索引设计

ALERT 结合 Radix Tree 和机器学习模型, 目标是在大量减少索引内存占用的同时, 使得整个索引的查询复杂度仍保持在常数级别. 图 1 描述了 ALERT 的整体结构. 索引采用两层结构, 自底向上构建: 下层将数据分段, 划分成多个 Segment; 上层的 Radix Tree 中存储了 Segment 的最大键和地址, 用于管理这些 Segment. 查找分为两步: 首先使用 Radix Tree 找到查找键 key 可能所在的 Segment, 然后在 Segment 内进行查找.

© 中国科学院软件研究所 http://www.jos.org.cn

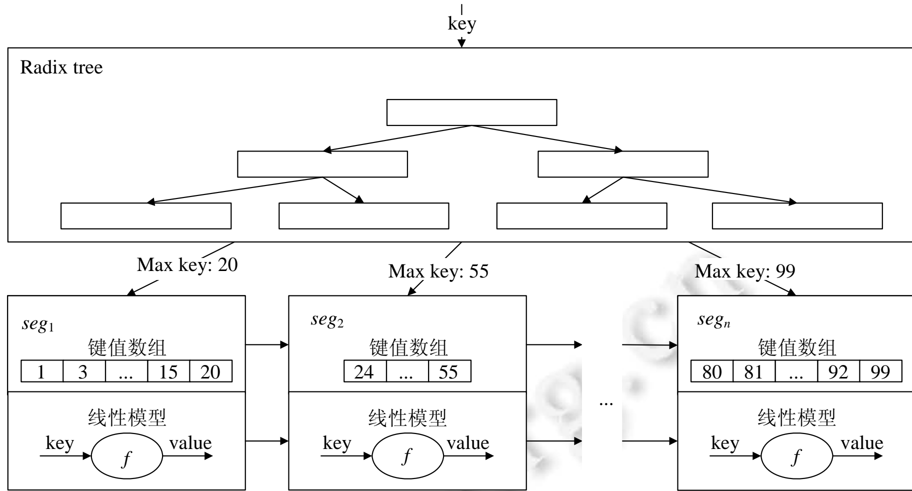


图 1　ALERT 结构

下面将分别对索引的两层结构进行详细介绍.

- 下层利用数据的分布将数据划分成多个不定长的 Segment. 每个 Segment 中的数据都训练一个机器学习模型, 通过模型预测数据的位置进行查询, 具有常数的时间复杂度, 而与 Segment 内的数据量无关. 不定长 Segment 具有更好的灵活性, 这也是减少总的 Segment 数量、进而减少索引内存占用的关键. 不同的 Segment 之间组织为链表结构, 相对有序, 使得范围查询也可以非常高效;
- 上层使用 Radix Tree 来管理 Segment. 假设第 i 个 Segment 中的最大键为 max_key_i, Segment 的地址为 seg_i, 则在 RadixTree 中存储形如$\langle max_key_i,seg_i\rangle$的一系列键值对. 上层选用 Radix Tree 有两个原因: 首先, Radix Tree 查询插入等操作的时间复杂度只与键长有关, 且 Segment 中的机器学习模型适用的数值类型键长较短, 从而对下层 Segment 的管理开销可以保持在较小的常数级别; 其次, 下层 Segment 的数量相对于原始数据量减少, 只索引 Segment 的情况下, Radix Tree 的大小降低, 从而对缓存更加友好, 具有更高的查询效率.

Radix Tree 相关研究较多, 对于 Radix Tree 的内部节点具有许多稀疏分布的分支的问题, 我们可以利用已有的相关工作给出的解决方案, 如 ART, 这里不作过多介绍. 下面重点介绍 ALERT 下层 Segment 的模型表示和分段算法, 以及如何结合下层的 Segment 和上层的 Radix Tree, 实现高效的批量加载、点查询和范围查询等操作.

2.1 Segment表示

Segment 的设计有两个目标: 利用数据分布实现高效的 Segment 内部搜索; 减少 Segment 的数量从而降低索引内存占用. 下面将首先讨论使用何种机器学习模型学习 Segment 内的数据分布, 然后使用一种基于贪心策略的算法尽可能地减少 Segment 的数量.

为了将机器学习模型引入 Segment 的设计之中, 需要解决两个问题.

- 选取什么模型来学习数据的分布. 模型的选取需要兼顾准确率和训练开销, 最近, 一些学习型索引的工作[1,4,17,19,22,23]表明: 使用一系列简单的线性模型, 如线性插值, 可以高效地建模数据分布;
- 如何处理模型的误差, 保证有界性. 模型的误差需要通过局部搜索来修正, 而局部搜索的效率取决于误差的大小. 常用的误差衡量准则是均方误差, 但均方误差并不能保证误差界, 可能会导致一个范围很大的局部搜索. 更好的选择是使用最大误差, 保证模型的预测值与真实分布的差值都在一个最大误差界内, 使得局部搜索的距离不会超过最大误差范围, 从而保证索引查找性能在最坏情况下的有界性.

© 中国科学院软件研究所 http://www.jos.org.cn

综合以上的讨论, 选择使用满足最大误差界的分段线性插值模型[24]来建模数据的分布. 下面给出使用满足最大误差的分段线性插值模型建模数据累积分布的形式化定义.

给定一组包含 m 个不同键的有序集合 $U=\{x_1,x_2,\dots,x_m\}$, 令 A 为一个大小为 $n(n\geqslant m)$, 有序存储 U 中键(允许重复)的数组. 对于集合 U 中的每一个键 x_i, 使用 $p_i=rank(x_i)$表示有序数组 A 中小于 x_i 的键的数量, 即 x_i 在数组中的第 1 个起始位置的下标, 令 $d_i=(x_i,p_i)$, 构造训练集 $D=\{d_1,d_2,\dots,d_m\}$. 定义 $S[i,j]=\{d_i,d_{i+1},\dots,d_j\}$, 给定最大误差界$\varepsilon$, 则目标是在满足最大误差界的前提下, 将 D 分成尽可能少的 k 个 Segment $S[i_1,i_2-1],S[i_2,i_3-1],\dots,S[i_k,i_{k+1}-1]$, 其中, $i_1=1$, $i_{k+1}-1=m$. 任意 Segment $S[i_j,i_{j+1}-1]$, $1\leqslant j\leqslant k$ 中的数据, 都可以用一个线性插值模型 $f_j(x)=a\cdot x+b$ 来近似, 并且满足最大误差:

$$|f_j(x)-p_i|\leqslant\varepsilon,\ i\in[i_j,i_{j+1}-1] \tag{1}$$

满足上述条件的线性插值模型可以用作对 Segment $S[i_j,i_{j+1}-1]$的查找, 并且模型的预测误差不超过ε, 从而保证了 Segment 内查找的最大误差界.

对于每一个 Segment, 可以看作一个五元组$\langle keys,values,slope,pre_segment,next_segment\rangle$, 其中, *keys* 和 *values* 是存储数据的键值数组, *slope* 是模型参数, *pre_segment* 指向上一个 Segment 的指针, *next_segment* 是指向下一个 Segment 的指针. 键值在数组中从 0 位置开始有序存放. 由于一个满足最大误差界的线性插值模型 $f(key)=slope\times key$ 便可近似 Segment 键值数组中数据的累计分布, 因此模型参数只有一个线性插值模型的斜率 *slope*. 通过模型 $f(key)$, 可以预测给定键在键值数组中的下标 *pos*, 从而只需在预测值附近$[pos-\varepsilon,pos+\varepsilon]$的较小范围内进行局部搜索来修正误差, 从而保证了常量的搜索时间.

2.2 分段划分算法

下面讨论如何在满足线性插值模型最大误差界的前提下, 将数据分成尽可能少的 Segment, 从而降低上层 Radix Tree 的大小和管理开销. 对于这个问题有很多相关研究, 这里使用 Liu 等人提出的一种称作可行空间窗口(feasible space window, FSW)的线性算法[25]. FSW 具有线性的时间复杂度和常量的空间开销, 其高效性和可行性也已在学习型索引[4,17]中得到了印证. FSW 使用贪心的思想, 在满足数据点误差边界的情况下, 尽可能找到最远处的划分点, 使得每一个 Segment 尽可能包含更多的数据点, 从而减少 Segment 的数量. 图 2 中, 横坐标为键的大小, 纵坐标为键的存储位置, 不同的键及其存储位置构成了坐标系中若干个不同的点 $d_i=(x_i,p_i)$.

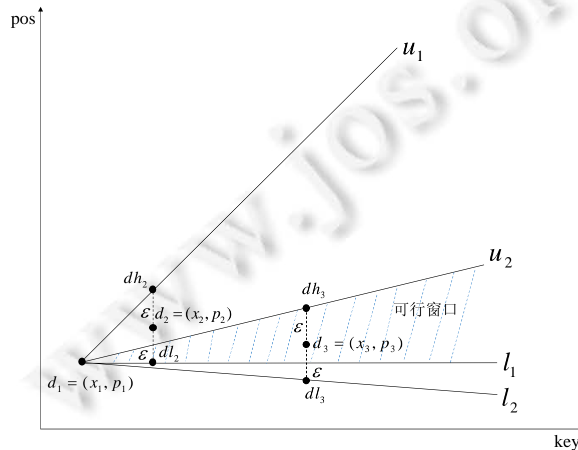


图 2 FSW 分段算法

下面结合图 2, 示意说明 FSW 算法的过程: 设最大误差界为ε, 令 $dh_i=(x_i,p_i+\varepsilon)$, $dl_i=(x_i,p_i-\varepsilon)$, 选择数据的第 1 个点 d_1 作为起始点, 依次扫描数据, 当第 2 个点 d_2 加入时, 由 d_1 和 dh_2 的连线 u_1 表示可行窗口的上界, 由 d_1 和 dl_2 的连线 l_1 表示可行窗口的下界, 二者中间包裹的区域称为可行窗口. 随着新的数据点的加入, 形成的上界 u_i 或下界 l_i 在可行窗口内, 则可行窗口也会进行收缩. 如当第 3 个点 d_3 加入时, 构造连线 u_2 和 l_2, 此时可

© 中国科学院软件研究所 http://www.jos.org.cn

行窗口收缩至 u_2 和 l_1 之间的区域. 当新加入的数据点 d_e 落在可行窗口外时, 说明之前加入的点至少存在一个违反最大误差, 此时使用起始点至 d_e 之前的点构建一个新的 Segment, 并将 d_e 作为新的起始点, 继续构建下一个 Segment, 直至扫描完全部数据.

3 索引操作

3.1 批量加载

批量加载操作用于构建或重建索引, ALERT 采用自底向上的方式批量加载来构建索引. 批量加载之前, 需要保证要加载的数据是有序的. 首先使用 FSW 分段算法构建 Segment 的集合, 将数据划分至不同 Segment 的键值数组中存储, 然后在每个 Segment 中保存线性插值模型的参数; 将每个 Segment 中最大的键和指向 Segment 的指针构成键值对, 插入到上层 Radix Tree 中进行管理, 并在下层将 Segment 组织成链表的形式.

批量加载算法的时间复杂度为 O(n), 这得益于 FSW 分段算法线性的时间复杂度和 Radix Tree 插入时只与键长有关的常量时间复杂度, 虽然构建过程中需要将数据拷贝至 Segment 的键值数组中, 但对于每一个元素, 批量加载仍只有常数的均摊开销, 因而批量加载算法具有较高的构建效率.

3.2 点查询

给定一个查找键 $keymax_key_i$, 点查询操作分为两步.

- 第 1 步, 在 Radix Tree 中查询 key 可能所在的 Segment. 对于查找键 key, 在 Radix Tree 查找键 max_key_i, 使得 $max_key_{i-1}<key\leqslant max_key_i$. max_key_i 为 key 在 Radix Tree 中第 1 个大于等于 key 的键, 对应的 Segment 即 key 可能所在的 Segment. 需要注意: 在 Radix Tree 中查询时, 需要将 key 转换成可比较的二进制字节数组形式, 如无符号长整型的 key, 可视为长度为 8 的字节数组;
- 第 2 步, 得到查找键 key 可能所在的 Segment 之后, 需要在 Segment 内搜索. 首先使用模型得到查找键的预测位置 $pred=(key-keys[0])\times slope$, 然后使用局部搜索修正模型预测的误差. 由于模型的预测位置与真实位置的最大误差不超过ε, 因此只需搜索预测位置附近的范围$[pred-\varepsilon,pred+\varepsilon]$.

查找键 key 的可比较二进制字节数组长度为 k, 则 Radix Tree 查询 key 所在 Segment 的时间复杂度为 O(k). Segment 内部搜索的时间复杂度取决于内部键值数组的搜索, 如果使用二分查找, 则是一个只与分段最大误差有关的常量 O($\log\varepsilon$). 综上, 点查询的时间复杂度为 O($k+\log\varepsilon$), 具有常量的查询时间.

3.3 范围查询

范围查询分为两步: 第 1 步类似点查询, 首先在 Radix Tree 中查询起始键可能所在的 Segment, 然后在此 Segment 中找到第 1 个不小于起始键的位置; 第 2 步进行遍历, 从 Segment 中第 1 个不小于起始键的位置开始, 直至到大于或者等于终止键. 由于 Segment 之间在下层组织成了链表形式, 如果当前 Segment 遍历结束,那么继续遍历下一个 Segment, 直至 Segment 中的键值落在查询区间之外或者所有 Segment 都已被遍历.

范围查询的开销由两部分组成: 一部分是查找第 1 个不小于起始键位置的开销, 此部分与点查询开销相同; 另一部分是遍历数据的开销, 这也是范围查询的主要开销所在, 并且查询的范围越大, 此部分开销占比越大. 由于数据在 Segment 内部键值数组中连续存储, Segment 之间组织成链表结构, 所以只需要从起始位置开始连续访问, 相对于 B 树和 Radix Tree 等范围查询时需要递归地遍历叶子节点的方式, 十分高效.

3.4 数据插入

不定长 Segment 带来了降低内存等好处, 但同时也给插入和更新带来了很大的挑战. 为了支持插入和更新, 有两种常用的方法: 原地更新(in-place)和使用单独缓冲区(delta buffer). In-place 通过分配冗余的空间容纳插入, 但会带来大量的键值移动; Delta Buffer 的问题是单个缓冲区会具有较多的键值数量, 会降低查询和插入的性能. 针对上面两种方法存在的问题, 本文设计了一种插入缓冲的方式, 有两个目标.

- 尽可能避免插入时数据移动带来的开销;

© 中国科学院软件研究所 http://www.jos.org.cn

- 减少缓冲区中键值数目过多对查询插入性能的不利影响.

首先分析使用 In-place 方式处理键值插入时的场景. 当新插入一个键值 *key* 时, 首先查找要插入的 Segment, 然后需要在 Segment 的键值数组中找到第 1 个不小于 *key* 的位置, 准备在此插入, 但此位置已有键值, 这种情况可以视作新插入的键值与原来存在的键值发生了冲突. 借鉴拉链法[26]解决哈希冲突的思想, 在 Segment 的键值数组中, 每一个冲突的位置均使用一个缓冲区容纳新的插入. 在任意一个 Segment 的键值数组 *seg.keys* 中, 其 *pos* 位置对应缓冲区中的键 *key*, 满足 $seg.keys[pos-1]<key\leqslant seg.keys[pos]$. 缓冲区默认采用链表实现, 以高效地应对频繁插入的场景. 插入缓冲并不维护链表中数据的有序性, 只需将新的键值插入到链表尾部.

这样的缓冲设计符合两个设计目标: 首先, 向缓冲区中插入键值不需要移动现有数据, 避免了键值数组中数据分布的改变, 因此不会降低模型精度, 使得键值数组中的搜索能够保证原有的性能; 其次, 每一个冲突的位置均使用一个缓冲区, 从而将新插入的键值分散存储在不同的缓冲区中, 缓解了单个缓冲区键值过多的开销, 并且也有利于后面更细粒度地感知查询的分布进行重组优化.

结合插入缓冲, ALERT 的插入操作十分高效, 查找要插入的 Segment 时间复杂度为 $O(k)$, 在 Segment 中查找要插入的缓冲区时间复杂度为 $O(\log\varepsilon)$, 将键值插入到缓冲区时间复杂度为 $O(1)$. 综上, 插入操作总的时间复杂度为 $O(k+\log\varepsilon)$, 保持在常数级别, 可以保证索引较高的插入效率.

4 自适应重组优化

随着数据的插入, 缓冲区中的数据不断积累, 导致落在缓冲区的查询效率较低, 因此需要对缓冲区中的数据进行重组优化. 最朴素的重组策略是: 当一个缓冲区中数据过多时, 将这个缓冲区所在 Segment 的所有缓冲区和键值数组中的数据合并, 通过分段算法重组为若干新的 Segment. 这种策略假设查询的访问模式是均匀的, 重组会给对 Segment 中所有新插入数据的查询带来收益. 然而在现实场景中, 某些数据的访问频率要高于其他数据, 并且在不久的将来更有可能被再次访问, 而某些数据很少被查询涉及. 上述策略在缺乏查询工作负载信息的情况下进行重组优化, 具有盲目性, 如果后续查询不会访问重组涉及的数据, 会带来大量不必要的开销.

为了降低重组优化的开销, 本节提出了两种自适应重组优化方法, 分别为感知点查询和范围查询. 将重组优化视为查询的副作用, 将对缓冲区的重组优化推迟到查询执行时. 图 3 展示了结合自适应重组优化的索引插入和查询的流程, 每个 Segment 均有一个对应的插入缓冲, 插入缓冲被划分为无序区和有序区.

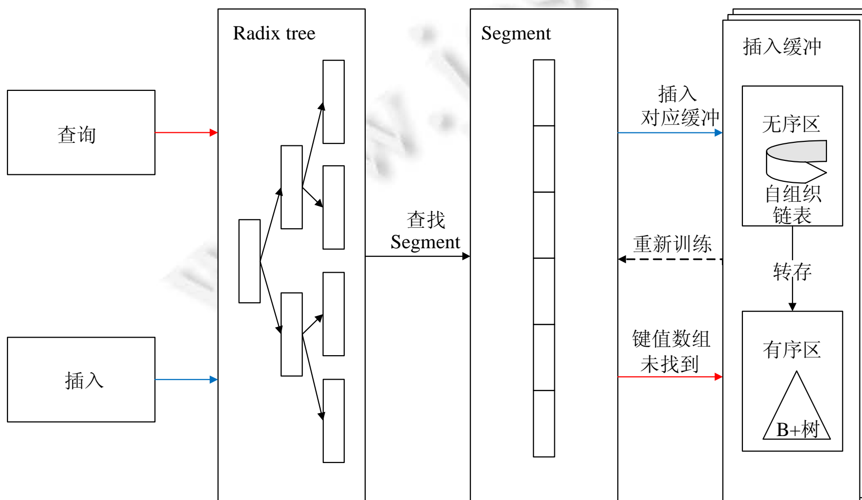


图 3 结合自适应重组优化的索引框架

查询和插入首先使用第 3.2 节中所描述点查询算法的第一步, 在 Radix Tree 中找到要插入的键对应的 Segment. 对于插入, 直接插入到 Segment 对应缓冲的无序区; 对于查询, 先查找有序区然后查找无序区, 并在查询过程中执行重组优化.

4.1 点查询自适应重组优化

为了在点查询过程中自适应地对缓冲区中的数据进行重组优化, 使用自组织链表[27]作为缓冲区的实现. 自组织链表最常用的自组织规则是 MTF (move-to-front)规则, 每当查找到一个节点, 将其移动到链表头部. MTF 规则基于最近被访问过的数据将来被访问的概率也更高的思想, 将热点数据移至链表头部, 在数据访问局部性较强的场景下, 比普通链表具有更短的平均访问距离. 但使用 MTF 规则会导致自组织链表单个操作的代价较高, 对于频繁访问的节点, 可能会在链表头部附近反复移动, 带来了许多不必要的开销.

为了解决上述问题, 设计了一种基于自适应窗口的 MTF 规则, 降低对于频繁访问节点单个操作的代价. 在链表头部维护一个大小为 k 的窗口, 窗口内维护的数据都是最近访问的前 k 个数据, 即链表中最可能被频繁访问的前 k 个节点. 查找时, 如果要查找的节点在窗口中, 那么直接返回, 而不需要移动; 如果不在窗口内, 那么仍需要按照 MTF 规则将此节点移动至链表头部, 此时, 原来窗口的最后一个节点被移出窗口. 窗口的存在, 避免了对频繁访问节点的移动, 减少了对查找的影响. 另外, 原始的 MTF 规则可视为窗口为 1 时的特殊情况.

下面确定窗口的大小 k, 即确定最可能被频繁访问节点的个数. 由于工作负载十分复杂多变, 人为很难根据经验设定 k 值. 因此, 采用基于指数衰减的方式来计算 k 值, 在线地更新窗口大小以适应工作负载的变化. 首先, 设置窗口初始大小为 0. 每次插入时, 线性扩张窗口, 将窗口大小加 1; 统计每次查找过程中找到目标节点时的访问距离 dis, 并使用指数衰减的方式更新窗口大小 $k_t=\alpha\cdot k_{t-1}+(1-\alpha)\cdot dis_t$, 其中, k_{t-1} 是第 $t-1$ 次查找后窗口的大小; k_t 是第 t 次查找后窗口的大小; dis_t 是第 t 次查找时的访问距离; α为衰减因子, 一般设置为 0.01~0.05[28], 较高的值会给予最近的查找更多的权重. 假设一个节点在真实分布中的访问概率为 p, 可以证明, 指数衰减的估计与真实分布的标准差为 $\sqrt{\alpha(1-p)/(2-p)}$ [29]. 使用指数衰减的方式简单高效, 准确率较高, 通过不断关注新的查找, 逐渐减少以往较久时间前查找的影响, 从而自适应查询分布的变化.

结合基于自适应窗口的 MTF 规则, 自组织链表在查找的同时进行重组优化. 当找到满足的节点时, 使用指数衰减更新窗口大小; 同时, 若访问距离大于窗口大小, 则将此节点移至链表头部. 相对于查找, 插入时首先线性扩张窗口的大小, 然后插入到链表的尾部即可. 综上所述, 通过使用基于自适应窗口 MTF 规则的自组织链表作为缓冲区的实现, 可以在不影响插入性能的同时, 使得缓冲区在进行点查询的同时重组数据, 从而逐渐自适应点查询的分布, 改善点查询的查询效率.

4.2 范围查询自适应重组优化

自组织链表通过感知点查询进行自适应重组优化, 但并不适用于范围查询, 有两个原因.

- 将满足范围查询条件的节点移动至链表头部会污染窗口, 降低点查询自适应重组优化的效果;
- 范围查询需要访问链表的所有节点, 因此移动节点到链表头部也不能优化范围查询.

为了解决上述问题, 提高范围查询效率, 在点查询自适应重组优化的基础上, 将缓冲区划分为无序区和有序区, 通过感知范围查询, 把无序区中的数据自适应合并至有序区, 在不影响插入性能和点查询自适应重组优化效果的前提下, 改善缓冲区的范围查询性能, 进一步地将范围查询频繁的 Segment 重新训练, 改善整个 Segment 中数据的查询性能.

无序区采用上节所述的自组织链表实现, 具有高效的插入性能, 并且可以自适应点查询的分布. 有序区采用 B+树实现, 能够较为高效地维护数据的有序性, 在数据量相对不大的情况下, 可以支持高效的范围查询. 无序区到有序区中的数据合并视为范围查询的副作用, 在进行范围查询的同时, 利用对无序区中满足查询条件的数据进行排序的时机, 将这些数据转换至有序区中存储, 在不影响无序区中自适应链表窗口的同时, 减少无序区中键值的数量, 减轻无序区中查询的压力, 从而使得整个缓冲区的组织结构逐渐趋于适应当前工作

负载，提升整个缓冲区的查询效率.

缓冲区使用链表和 B+树，相对于键值数组，具有更多的内存占用，并且不能够有效地利用数据的分布. 随着插入的进行, Segment 的缓冲区中存储的数据越来越多，这种情况将越来越严重，因此需要进行重新训练，将键值数组和缓冲区中的数据合并，使用分段算法将这些数据存储在若干个新 Segment 的键值数组中，从而降低索引的内存占用，提升后续查询的效率.

重新训练的开销来自于两个方面：一个是合并时排序的开销，另一个是分段算法的开销. 为了减少重新训练对索引的负面影响，需要确定重新训练的时机.

合并缓冲区和键值数组中的数据时，需要对缓冲区无序区中的数据进行排序. 为了减少此部分开销，需要尽可能地减少缓冲区中无序区数据的数量. 因此，重新训练需要满足的第 1 个条件为

$$\frac{\sum_{i=0}^{seg.size} seg.buffer[i].ordered_buffer.size}{\sum_{i=0}^{seg.size} seg.buffer[i].size} > threshold \tag{2}$$

即：在一个 Segment 缓冲区中有序区存储的键值总数达到缓冲区中所有键值总数的一定比例时，才考虑进行重新训练，从而减小对无序数据排序的开销. 另外，由于范围查询是将无序区中数据合并至有序区的唯一途径，缓冲区中有序区键值数目较多，说明了对此 Segment 的范围查询较为频繁，而重新训练可以大大改善范围查询的效率，由此证明了此 Segment 具有足够重新训练的必要.

分段算法具有线性的开销，但重新训练需要涉及较多的数据，因此，重新训练需要满足的第 2 个条件为缓冲区中的键值总数目达到所在 Segment 中键值总数目的一定比例，使得每个键值可以均摊重新训练的开销.

5 实验评估

本文对 ALERT 进行了详细的实验，并与传统索引 B+树和 ART，以及学习型索引 PGM-index 和 ALEX，在真实数据集和合成数据集上进行了大量的对比. 所有实验都在一台使用 64 位 Ubuntu16.04 操作系统、内核版本为 4.4.0-131-generic 的服务器上进行，内存为 64 GB，使用型号为 Intel(R) Xeon(R) Silver 4114 CPU@ 2.20 GHz 的处理器. ALERT 使用 C++实现，上层的 Radix Tree 采用 ART. 实验过程中，将 ALERT 和 PGM-index 的分段最大误差设为 32, ALEX 的最大节点大小设置为 1 MB，其余索引均采用默认设置.

实验中使用 4 个数据集，包括 2 个真实数据集 face[30], osmc[31]和 2 个合成数据集 normal, uniform，这些数据集在现有学习型索引的研究工作中被广泛使用. face 是 Facebook 网站上的用户 ID, osmc 是公开地图 OpenStreetMap 上的 Cell ID, normal 是从参数为(0,2)的正态分布中采样生成, uniform 是一系列连续的无符号整数. 每个数据集均包含 2 亿个 64 位的无符号整数. 将每一个无符号整数视为一个键，然后为每一个键随机生成一个 8 字节的值，从而构成实验中所需要的键值对数据集. 对于一个给定的数据集，首先批量加载 2 亿键值初始化索引，然后执行 1 千万次查询或插入操作，然后报告操作的平均延迟.

5.1 构建性能

使用 2 亿有序键值批量加载索引，观察索引的构建性能. 图 4 展示了 6 种不同索引分别在 4 个数据集上的批量加载时间，时间越短，说明构建性能越好. ALERT 具有较高的构建效率，达到了与 B+树相近的速度. 在 4 个数据集上，与 ART 相比，构建时间平均降低了 73%；与静态学习型索引 PGM 相比，构建时间平均降低了 64%；与动态学习型索引 DPGM 和 ALEX 相比，构建时间平均降低了 81%. 这得益于索引使用的 FSW 分段算法只需遍历一次数据，具有线性的构建时间. 理论上，PGM-index 的分段算法也具有线性的时间复杂度，但其常数系数要高于 ALERT. ART 在构建过程中需要动态地扩张内部节点，在大量数据下，构建性能会急剧下降. 虽然 ALERT 上层也采用 ART，但因为只需要索引 Segment，数据量大大减少，降低了上层索引耗费的开销.由于 ALEX 需要在批量加载过程中递归地构建模型，其耗时远远高于其他 5 个索引. 另外可以发现, ALERT 和 ALEX 在分布较为简单的合成数据集上比在复杂的真实数据集上构建时间更短. 这也说明了学习型索引对于分布较为简单的数据构建性能更加高效，而复杂的分布则会影响学习型索引的构建性能.

© 中国科学院软件研究所 http://www.jos.org.cn

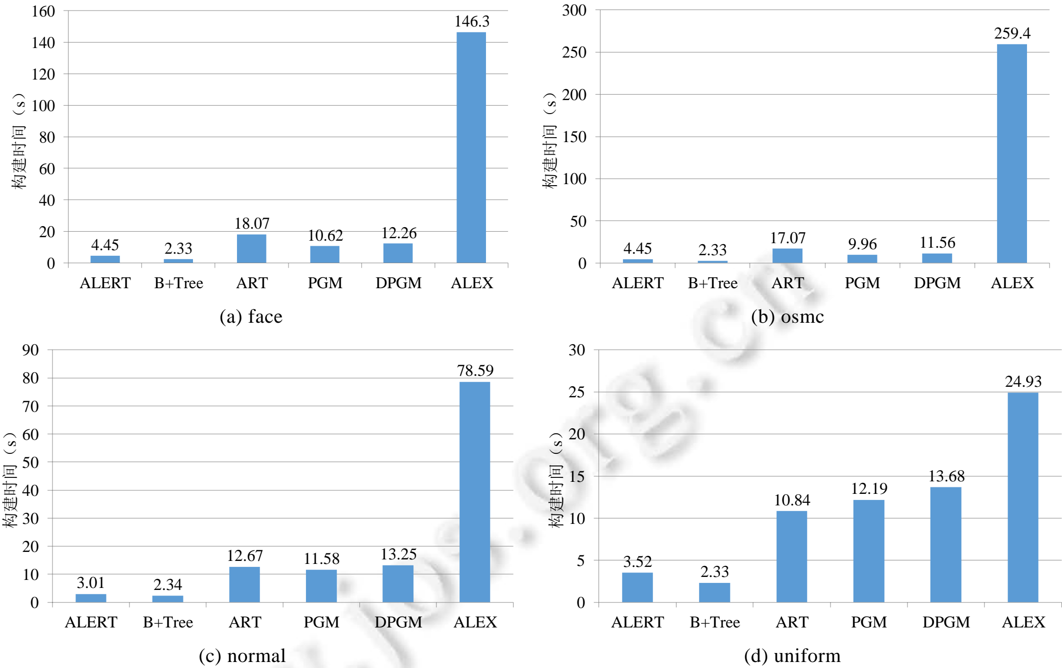


图 4 批量加载时间(越低越好)

5.2 索引大小

批量加载2亿键值之后, 比较每个索引大小. 索引大小使用内存占用来衡量, 各个索引大小的计算方式如下: B+树只在叶子节点中存储键值数据, 因此索引大小为所有内部节点之和; ART 的索引大小为所有节点之和减去所有键值的大小, 因为键值被编码进了节点; PGM 的索引大小为所有节点的总大小减去所有键值的大小; ALEX 的索引大小为索引使用的所有模型和元数据的大小总和; ALERT 的大小为上层 Radix Tree 的大小加上下层 Segment 的大小再减去所有键值的大小. 图 5 展示了 6 种不同索引在 4 个数据集上的索引大小. 在 4 个数据集上, ALERT 相对于传统索引, 减少了内存占用, 尤其在合成数据集 normal 和 uniform 上的大小几乎可以忽略不计, 即使在分布较为复杂的真实数据集 face 和 osmc 上, 相对于 B+树, 也分别降低了 54.2% 和 73.3%, 相对于 ART 分别降低了 95.8% 和 97.5%. 这也说明了学习型索引利用数据分布减少内存占用的优势.

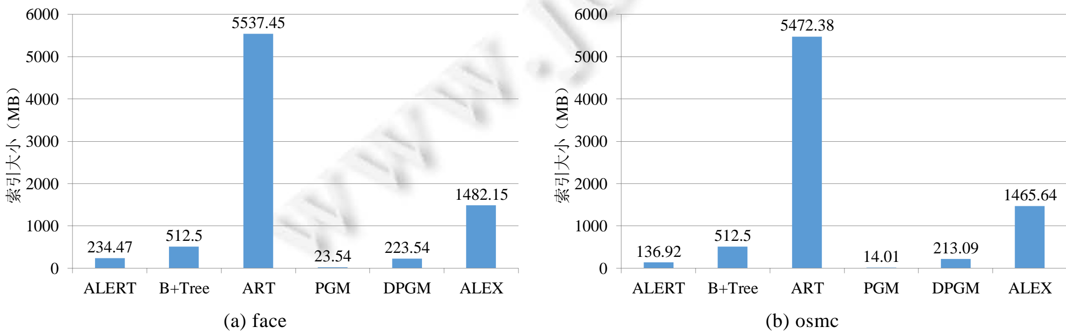
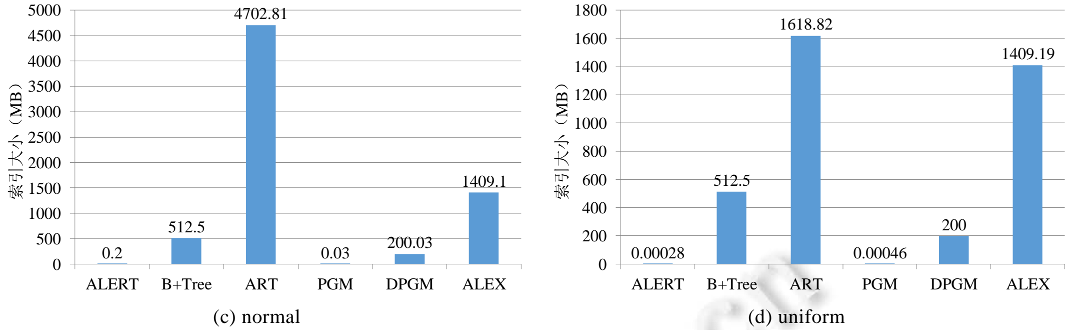


图 5 索引大小(越低越好)

© 中国科学院软件研究所 http://www.jos.org.cn



(c) normal (d) uniform

图 5 索引大小(越低越好)(续)

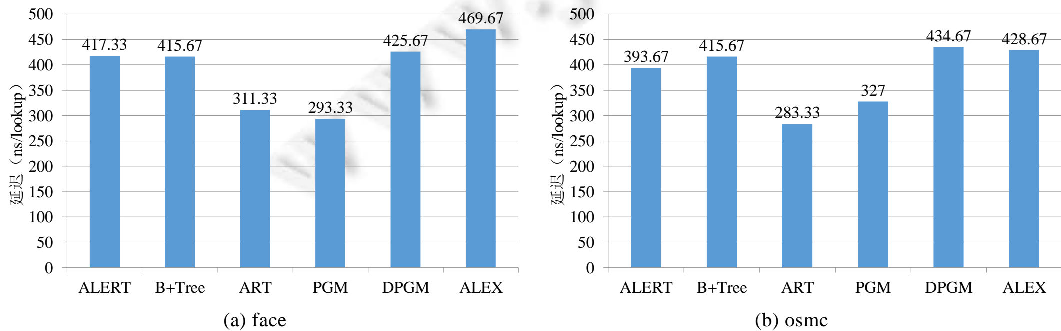
传统索引的内存占用要远大于学习型索引，尤其是 ART，虽然通过自适应节点相对于普通 Radix Tree 取得了较大改进，但仍具有相当大的内存占用. Leis 等人[11]也指出，ART 平均使用 8–52 字节来存储每一个键，这也与实验结果相符. 与学习型索引 DPGM 和 ALEX 相比，ALERT 的内存占用平均降低了 75%. 虽然 PGM 具有更少的内存占用，但其动态版本 DPGM 则需要更多的内存；ALEX 使用了 Gapped Array 作为节点的实现，具有冗余的空间，所以也具有相对较大的内存占用.

5.3 查询性能

查询操作包括点查询和范围查询，二者的性能分别使用一批查询操作的平均延迟来衡量. 查询操作的选取满足一定的分布，查找键的选取使用 Zipfian 分布，从而关注某些热点记录；范围查询要扫描键的数量使用均匀分布，从 0–100 的范围内随机选择. 在批量加载构建好的索引上分别执行 1 千万次点查询和 1 千万次范围查询操作，观察 6 种不同索引在 4 个数据集上两种查询操作的平均延迟. 下面分别对 ALERT 的点查询性能和范围查询性能进行讨论.

点查询的实验结果如图 6 所示，在 normal 和 uniform 数据集上，学习型索引可以很好地学习数据的分布，相对于 B+树和 ART，ALERT 将延迟最大分别降低了 78.1% 和 39.3%. 即使在分布十分复杂的 face 和 osmc 数据集上，ALERT 也达到了与 B+树相近的性能. 在学习型索引之中，ALERT 的点查询性能相对于动态学习型索引 DPGM 和 ALEX 在 4 个数据集上均具有优势，但差于静态学习型索引 PGM. 这是因为 PGM 通过递归地构建线性分段模型，可以很好地学习数据中非线性的分布；而动态学习型索引，为了性能往往采用简单的模型和更为复杂的数据结构，因此相对于静态学习型索引牺牲了一定的查询性能.

范围查询的实验结果如图 7 所示，ALERT 在 4 个数据集上均取得了优异的范围查询性能，尤其在 normal 和 uniform 上表现最优.



(a) face (b) osmc

图 6 点查询延迟(越低越好)

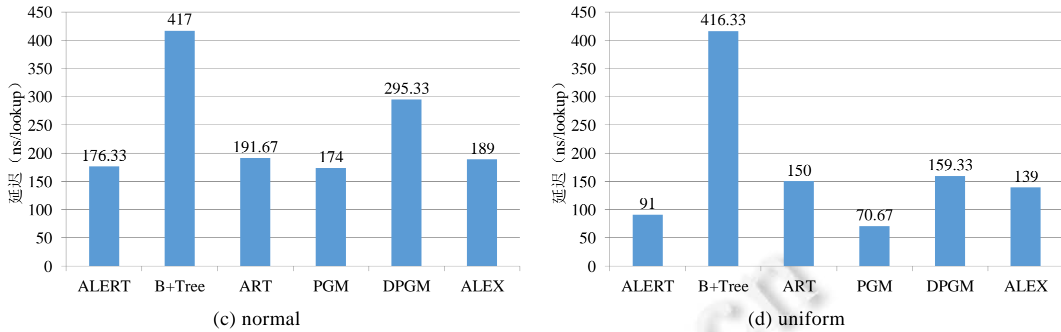


图 6　点查询延迟(越低越好)(续)

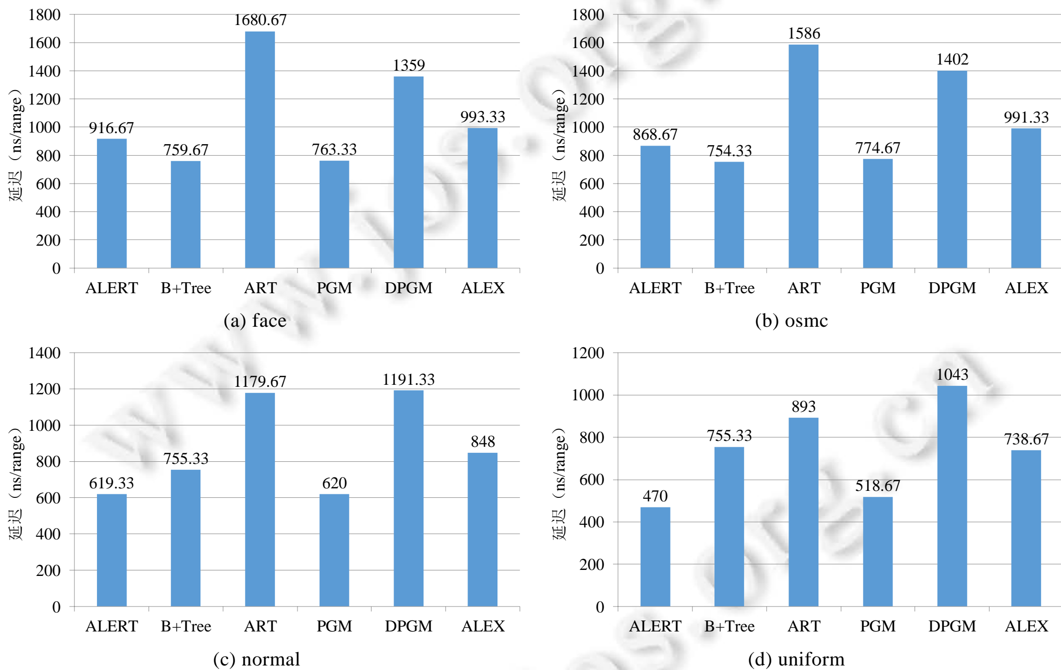


图 7　范围查询延迟(越低越好)

与点查询不同, 范围查询的性能主要取决于查询的范围, B+树、PGM 和 ALERT 使用数组存储键值, 当找到第 1 个满足查询条件的键值之后, 只需连续地访问下层的键值数组, 因此范围查询的性能较好. DPGM 由于将键值分散存储于多个静态 PGM-index 之中, 访问时需要遍历多个静态 PGM-index 并合并结果, 因此相对于 PGM, 性能大大降低. ALEX 将键值存储于 Gapped Array, 访问时需要跳过间隙, 也影响了性能. ART 范围查询时需要递归遍历叶子节点, 因此性能最差. ALERT 在 4 个数据集上的范围查询延迟相对于 ART, DPGM 和 ALEX 最大分别降低了 47.5%, 54.9% 和 36.4%. 可以预见: 随着查询范围的增大, ALERT 可以连续访问下层键值数组带来的优势, 将使得这种差距越来越大.

5.4　插入性能

在批量加载构建好的索引上运行执行 1 千万次插入操作, 插入操作使用均匀分布从数据集中选取. 因为静态索引 PGM 不支持插入, 因此观察剩余 5 种不同索引在 4 个数据集上插入操作的平均延迟. 实验结果如图 8 所示: ALERT 达到了与传统索引 B+树和 ART 相近的性能, 在 normal 和 uniform 数据集上, 甚至优于传统索引. 与学习型索引相比, ALERT 在 4 个数据集上的插入延迟相对于 DPGM 和 ALEX 平均降低了 50%. 这是由

© 中国科学院软件研究所 http://www.jos.org.cn

于ALEX需要进行节点内数据的移动甚至节点的分裂和递归插入, DPGM则需要对若干个已满的静态PGM进行合并和重新训练, 均具有较大的开销. 而ALERT只需找到对应的缓冲区, 将键值追加到缓冲区的无序区中, 通过将维护索引中数据有序和重新训练等重组工作视为查询的副作用, 推迟执行, 保证了较高的插入效率.

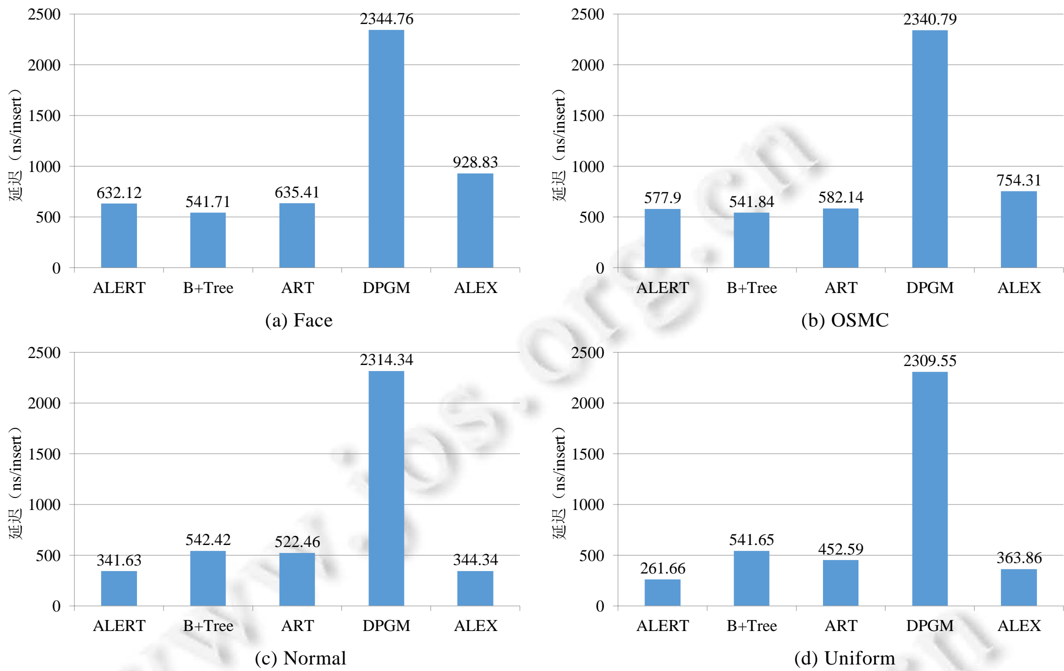


图8 插入延迟(越低越好)

5.5 分段最大误差的影响

分段最大误差影响ALERT中分段的一个重要参数, 它影响了索引占用内存的大小和查询的延迟. 设置不同的分段最大误差大小, 在face数据集上使用2亿个数据构建索引, 并进行1千万次点查询操作, 观察不同分段最大误差下索引的表现情况.

图9展示了不同分段最大误差下的查询延迟以及索引大小.

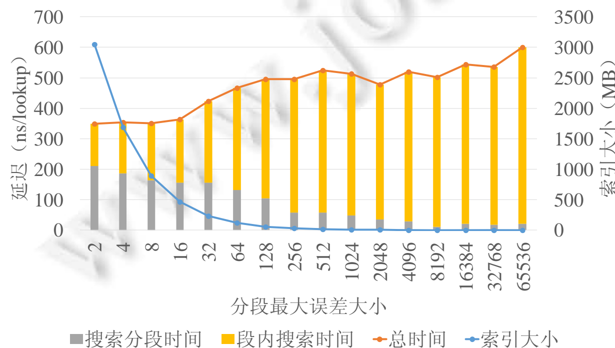


图9 不同最大误差下的查询延迟及索引大小

增大分段最大误差可以减少Segment数量, 使得索引具有更小的上层Radix Tree, 从而也降低了索引的内存占用. 点查询操作延迟的总时间分为两部分: 一部分是在Radix Tree中搜索Segment的时间, 另一部分是Segment内进行搜索修正模型误差的时间. 随着分段最大误差大小的增加, 搜索Segment的时间占总时间的比

例越来越小. 这是因为较小的 Radix Tree 更容易被缓存, 具有更高的搜索效率. 但与此同时, 由于 Segment 内局部搜索的范围与分段最大误差的大小有关, 增大分段最大误差会使得此部分开销越来越大, 从而造成了点查询操作延迟总时间上的增长.

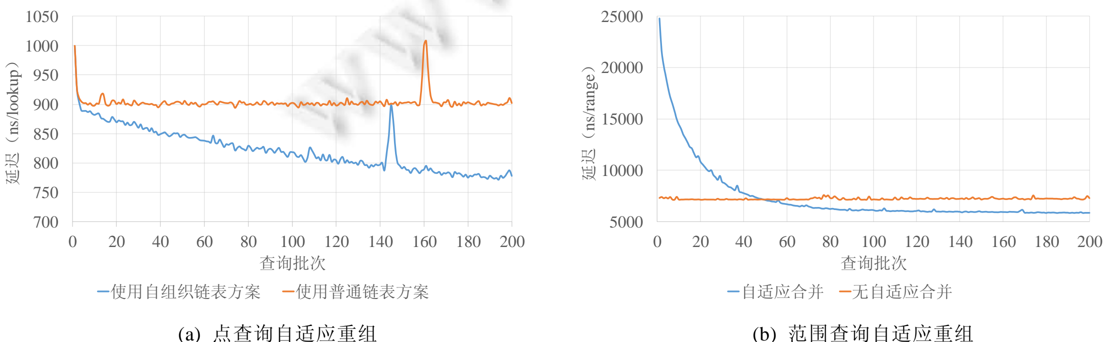
下面给分段最大误差的设置提供一些指导准则, 较大的分段最大误差使索引具有更低的内存占用, 但会对查询性能造成损耗. 也就是说, 牺牲一定的查询性能可以换取更低的内存占用. 例如, 将分段最大误差大小由 2 调至 16, 仅增大了的 4.3% 查询延迟, 却可以带来 84.7% 的索引占用内存大小的降低. 根据这个准则, 可以很方便地通过调节分段最大误差的大小对索引的内存占用和查询延迟三者进行权衡. 但需要注意的是, 分段最大误差大小的设置并不是越大越好. 从图 9 可以看到: 此场景下, 当分段最大误差超过 128 时, 索引的内存占用大小基本不再降低, 而查询延迟却在增长. 因此, 可以将分段最大误差的大小设置在 16–128 之间.

5.6 自适应重组优化效果

为了观察缓冲区点查询和范围查询自适应重组优化的效果, 首先对包含 2 亿数据的 face 数据集进行下采样, 得到 2 百万数据; 然后批量加载这 2 百万数据构建索引, 并将剩余数据随机插入, 构造缓冲区中具有大量新插入键值的场景. 接下来, 分 200 个批次分别进行 1 千万次点查询和 1 千万次范围查询, 观察各批次查询之间平均延迟的变化趋势. 查询的键值使用 Zipfian 分布从新插入的数据中选取, 从而使得查询都落在缓冲区中, 范围查询的范围从 0–1000 随机选取. 下面分别对点查询和范围查询自适应重组优化的效果进行分析.

图 10(a)展示了分别使用自组织链表和普通链表时, 点查询平均延迟的变化趋势. 起初, 二者都具有较高的延迟, 这是由于正在预热操作系统缓存. 从第 2 个批次开始, 使用普通链表方案的查询延迟逐渐趋向平稳, 并在 900 ns 左右震荡; 而使用自组织链表的方案通过自适应查询的分布, 延迟逐渐降低. 在第 4 个批次之后, 使用自组织链表方案的查询延迟开始低于使用普通链表的方案. 第 180 个批次之后, 自组织链表的自适应窗口逐渐趋于稳定, 查询延迟稳定在 770 ns 附近, 相较于普通链表方案, 降低了约 14.4% 的延迟. 两种方案的查询延迟在 140–160 批次均出现波动, 这可能是由于不同批次查询分布的变化, 导致索引查询过程中出现缓存失效造成的.

图 10(b)展示了使用自适应合并和不使用自适应合并的情况下, 范围查询平均延迟的趋势变化. 起初, 范围查询需要将无序区中满足查询范围的数据合并至有序区, 从而具有较大的延迟. 但随着缓冲区中数据的不断重组和合并, 从第 50 个批次开始, 使用自适应合并方案的延迟开始低于不使用自适应合并的方案. 最终, 从第 110 个批次开始稳定在 6 050 ns 附近. 而不使用自适应合并方案的延迟则一直在 7 150 附近波动, 相较于不使用自适应合并的方案, 使用自适应合并的方案降低了 15.4% 的延迟. 不使用自适应合并的方案始终需要遍历所涉及无序区中的所有数据, 而使用自适应合并的方案仅需要在第 1 次遍历无序区时, 将范围查询涉及合并至有序区, 这是使用自适应合并方案稳定后优于不使用自适应合并的关键所在. 此外, 自适应合并只需对查询涉及的数据进行重组, 具有较低的开销. 可以看到: 在自适应合并的过程中, 查询的延迟呈现指数式降低, 说明了自适应合并具有较快的收敛速度, 可以很快自适应查询的分布.



(a) 点查询自适应重组 (b) 范围查询自适应重组

图 10 自适应重组优化效果

6 总结与展望

本文设计了一种基于 Radix Tree 的工作负载自适应学习型索引 ALERT, 旨在利用数据的分布减少索引的内存占用, 保证相对高效的查询性能. 通过使用一种高效的插入缓冲, 降低数据插入更新的成本, 解决学习型索引插入更新支持较弱的问题. 同时, 使用两种自适应重组优化方法, 通过感知点查询和范围查询的分布, 自适应工作负载, 以较低的代价降低了插入对于索引性能的影响. 但本文工作仍存在以下不足之处: ALERT 目前只支持单线程下的查询和插入更新, 为 ALERT 提供并发支持具有重要意义; 目前, 学习型索引支持的键类型受限于定长的数值类型, 如何支持索引不定长的字符串类型, 也是未来考虑优化的一个方向.

References:

[1] Kraska T, Beutel A, Chi EH, *et al*. The case for learned index structures. In: Proc. of the 2018 Int'l Conf. on Management of Data. 2018. 489−504.

[2] Chai MK, Fan J, Du XY. Learnable database systems: Challenges and opportunities. Ruan Jian Xue Bao/Journal of Software, 2020, 31(3): 806−830 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5908.htm [doi: 10.13328/j.cnki.jos.005908]

[3] Hadian A, Heinis T. Considerations for handling updates in learned index structures. In: Proc. of the 2nd Int'l Workshop on Exploiting Artificial Intelligence Techniques for Data Management. 2019. 1−4.

[4] Galakatos A, Markovitch M, Binnig C, *et al*. Fiting-tree: A data-aware index structure. In: Proc. of the 2019 Int'l Conf. on Management of Data. 2019. 1189−1206.

[5] Ding J, Minhas UF, Yu J, *et al*. ALEX: An updatable adaptive learned index. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. 2020. 969−984.

[6] Li X, Li J, Wang X. ASLM: Adaptive single layer model for learned index. In: Proc. of the Int'l Conf. on Database Systems for Advanced Applications. Springer, 2019. 80−95.

[7] Gao YN, Ye JB, Yang NZ, *et al*. 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]

[8] Knuth DE. The Art of Computer Programming. Pearson Education, 1997, 3.

[9] Comer D. Ubiquitous B-tree. ACM Computing Surveys (CSUR), 1979, 11(2): 121−137.

[10] Bayer R, McCreight E. Organization and maintenance of large ordered indexes. In: Proc. of the Software Pioneers. Springer, 2002. 245−262.

[11] Rao J, Ross KA. Cache conscious indexing for decision-support in main memory. 1998.

[12] Rao J, Ross KA. Making B+-trees cache conscious in main memory. In: Proc. of the 2000 ACM SIGMOD Int'l Conf. on Management of Data. 2000. 475−486.

[13] Kim C, Chhugani J, Satish N, *et al*. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In: Proc. of the 2010 ACM SIGMOD Int'l Conf. on Management of data. 2010. 339−350.

[14] Leis V, Kemper A, Neumann T. The adaptive radix tree: artful indexing for main-memory databases. In: Proc. of the 2013 IEEE 29th Int'l Conf. on Data Engineering (ICDE). IEEE, 2013. 38−49.

[15] Kemper A, Neumann T. HyPer: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In: Proc. of the 2011 IEEE 27th Int'l Conf. on Data Engineering. IEEE, 2011. 195−206.

[16] Raasveldt M, Mühleisen H. DuckDB: An embeddable analytical database. In: Proc. of the 2019 Int'l Conf. on Management of Data. 2019. 1981−1984.

[17] Kipf A, Marcus R, Van Renen A, *et al*. RadixSpline: A single-pass learned index. In: Proc. of the 3rd Int'l Workshop on Exploiting Artificial Intelligence Techniques for Data Management. 2020. 1−5.

[18] Bender MA, Farach-Colton M, Mosteiro MA. Insertion sort is O(*n*log*n*). Theory of Computing Systems, Springer, 2006, 39(3): 391−397.

[19] 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(10): 1162−1175.

[20] O'Neil P, Cheng E, Gawlick D, *et al*. The log-structured merge-tree (LSM-tree). Acta Informatica, 1996, 33(4): 351−385.

© 中国科学院软件研究所 http://www.jos.org.cn

[21] Overmars MH. The Design of Dynamic Data Structures. Springer Science & Business Media, 1987. 156.

[22] Kipf A, Marcus R, Van Renen A, *et al*. SOSD: A benchmark for learned indexes. arXiv preprint arXiv: 1911.13014, 2019.

[23] Marcus R, Kipf A, Van Renen A, *et al*. Benchmarking learned indexes. arXiv preprint arXiv: 2006.12804, 2020.

[24] O'Rourke J. An on-line algorithm for fitting straight lines between data ranges. Communications of the ACM, 1981, 24(9): 574–578.

[25] Liu X, Lin Z, Wang H. Novel online methods for time series segmentation. IEEE Trans. on Knowledge and Data Engineering, 2008, 20(12): 1616–1626.

[26] Cormen TH, Leiserson CE, Rivest RL, *et al*. Introduction to Algorithms. MIT Press, 2009.

[27] McCabe J. On serial files with relocatable records. Operations Research, 1965, 13(4): 609 –618.

[28] Levandoski JJ, Larson PÅ, Stoica R. Identifying hot and cold data in main-memory databases. In: Proc. of the 29th IEEE Int'l Conf. on Data Engineering (ICDE). IEEE, 2013. 26–37.

[29] Movellan JR. A quickie on exponential smoothing. https://inc.ucsd.edu/mplab/tutorials/ExpSmoothing.pdf

[30] Van Sandt P, Chronis Y, Patel JM. Efficiently searching in-memory sorted arrays: Revenge of the interpolation search? In: Proc. of the 2019 Int'l Conf. on Management of Data. 2019. 36–53.

[31] Pandey V, Kipf A, Neumann T, *et al*. How good are modern spatial analytics systems? Proc. of the VLDB Endowment, 2018, 11(11): 1661–1673.

附中文参考文献:

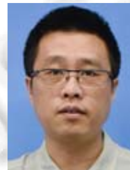
[2] 柴茗珂, 范举, 杜小勇. 学习式数据库系统: 挑战与机遇. 软件学报, 2020, 31(3): 806–830. http://www.jos.org.cn/1000-9825/5908.htm [doi: 10.13328/j.cnki.jos.005908]

[7] 高远宁, 叶金标, 杨念祖, 等. 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]

陈井爽(1996－), 男, 硕士, 主要研究领域为数据库系统.

江大伟(1982－), 男, 博士, 研究员, 博士生导师, 主要研究领域为分布式数据库技术, 云数据管理技术, 大数据管理技术, 区块链技术.

陈珂(1977－), 女, 博士, 副研究员, CCF 专业会员, 主要研究领域为数据库系统, 大数据技术, 隐私保护.



陈刚(1973－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为数据库系统, 大数据技术, 数据智能计算.

寿黎但(1974－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为数据库系统, 数据智能技术, 数据挖掘.

© 中国科学院软件研究所 http://www.jos.org.cn