软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2020,31(3):620−633 [doi: 10.13328/j.cnki.jos.005910]

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

基于中间层的可扩展学习索引技术*

高远宁[1,2], 叶金标[1,2], 杨念祖[1,2], 高晓泓[1,2], 陈贵海[1,2]

[1](上海市可扩展计算与系统重点实验室,上海 200240)

[2](上海交通大学 计算机科学与工程系,上海 200240)

通讯作者: 高晓泓, E-mail: gao-xf@cs.sjtu.edu.cn

摘 要: 在大数据与云计算时代,数据访问速度是衡量大规模存储系统性能的一个重要指标.因此,如何设计一种轻量、高效的数据索引结构,从而满足系统高吞吐率、低内存占用的需求,是当前数据库领域的研究热点之一.Kraska 等人提出使用机器学习模型代替传统的 B 树索引,并在真实数据集上取得了不错的效果,但其提出的模型假设工作负载是静态的、只读的,对于索引更新问题没有提出很好的解决办法.提出了基于中间层的可扩展的学习索引模型 Dabble,用来解决索引更新引发的模型重训练问题.首先,Dabble 模型利用 *K*-Means 聚类算法将数据集划分为 *K* 个区域,并训练 *K* 个神经网络分别学习不同区域的数据分布.在模型训练阶段,创新性地把数据的访问热点信息融入到神经网络中,从而提高模型对热点数据的预测精度.在数据插入时,借鉴了 LSM 树延迟更新的思想,提高了数据写入速度.在索引更新阶段,提出一种基于中间层的机制将模型解耦,从而缓解由于数据插入带来的模型更新问题.分别在 Lognormal 数据集以及 Weblogs 数据集上进行实验验证,结果表明,与当前先进的方法相比,Dabble 模型在查询以及索引更新方面都取得了非常好的效果.

关键词: 学习索引;聚类;神经网络;动态更新

中图法分类号: TP18

中文引用格式: 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

英文引用格式: 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). http://www.jos.org.cn/1000-9825/5910.htm

Middle Layer Based Scalable Learned Index Scheme

GAO Yuan-Ning[1,2], YE Jin-Biao[1,2], YANG Nian-Zu[1,2], GAO Xiao-Feng[1,2], CHEN Gui-Hai[1,2]

[1](Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai 200240, China)

[2](Department of Computer Science and Engineering, Shanghai JiaoTong University, Shanghai 200240, China)

Abstract: In the era of big data and cloud computing, efficient data access is an important metric to measure the performance of a large-scale storage system. Therefore, design a lightweight and efficient index structure, which can meet the system's demand for high throughput and low memory footprint, is one of the research hotspots in the current database field. Recently, Kraska, *et al* proposed to use the machine learning models instead of traditional B-tree indexes, and remarkable results are achieved on real data sets. However, the

* 基金项目: 国家重点研发计划(2018YFB1004700); 国家自然科学基金(61872238, 61972254, 61832005); 上海市科技创新行动计划(17510740200); CCF-华为数据库创新研究计划(CCF-Huawei DBIR2019002A)

Foundation item: National Key Research and Development Program of China (2018YFB1004700); National Natural Science Foundation of China (61872238, 61972254, 61832005); Shanghai Science and Technology Fund (17510740200); CCF-Huawei Database System Innovation Research Plan (CCF-Huawei DBIR2019002A)

本文由人工智能赋能的数据管理、分析与系统专刊特约编辑李战怀教授、于戈教授和杨晓春教授推荐.

收稿时间: 2019-07-20; 修改时间: 2019-09-10, 2019-11-25; 采用时间: 2019-12-18; jos 在线出版时间: 2020-01-10

CNKI 网络优先出版: 2020-01-10 13:34:55, http://kns.cnki.net/kcms/detail/11.2560.TP.20200110.1334.010.html

proposed model assumes that the workload is static and read-only, failing to handle the index update problem. This study proposes Dabble, a middle layer based scalable learning index model, which is used to mitigate the index update problem. Dabble first uses *K*-means algorithm to divide the data set into *K* regions, and trains *K* neural networks to learn the data distribution of different regions. During the training phase, it innovatively integrates the data access patterns into the neural network, which can improve the prediction accuracy of the model for hotspot data. For data insertion, it borrows the idea of LSM tree, i.e., delay update mechanism, which greatly improved the data writing speed. In the index update phase, a middle layer based mechanism is proposed for model decoupling, thus easing the problem of index updating cost. Dabble model is evaluated on two datasets, the Lognormal distribution dataset and the real-world Weblogs dataset. The experiment results demonstrate the effectiveness and efficiency of the proposed model compared with the state-of-the-art methods.

Key words: learned index; clustering; neural network; dynamic update

随着信息量爆发式地增长,如何高效快捷地对数据进行查询、更新,是衡量大规模存储系统的指标之一.因此,如何设计一种轻量、高效的索引结构,是提高数据库系统性能的关键.几十年来,研究者致力于各种索引结构的设计[1−6],包括红黑树、B 树、哈希和布隆过滤器等[1,7−9].研究表明,当今的商用数据库中,索引结构占据服务器内存的 55%[10].随着数据规模增大,主存不能容纳庞大的数据索引,需要将一部分索引放在外存中,从而增加了 I/O 操作数量[11].另外,传统索引的设计面向通用数据结构,并没有利用现实世界里的数据分布规律.例如:如果我们想要建立一个数据库存储和查询 100M 个以连续整数作为键值的记录,我们可以将键视为偏移量,从而将查询的时间复杂度由 B 树的 $O(\log N)$降低到为 $O(1)$;同时,索引的内存占用也从 $O(N)$减少到 $O(1)$.

针对传统索引结构的上述缺点,Kraska 等人提出了一种关注数据分布规律的索引结构——学习索引(learned index)[12].学习索引将索引本身看成机器学习模型,可以通过训练的方式得到一个自适应的模型,如图 1 所示,输入一个待查询的键,返回该键所在记录的位置.但机器学习模型的预测往往不是精确的,存在一定误差,因此需要在训练时记录模型的最大、最小误差,最终在误差范围内采用二分搜索确定记录的准确位置.

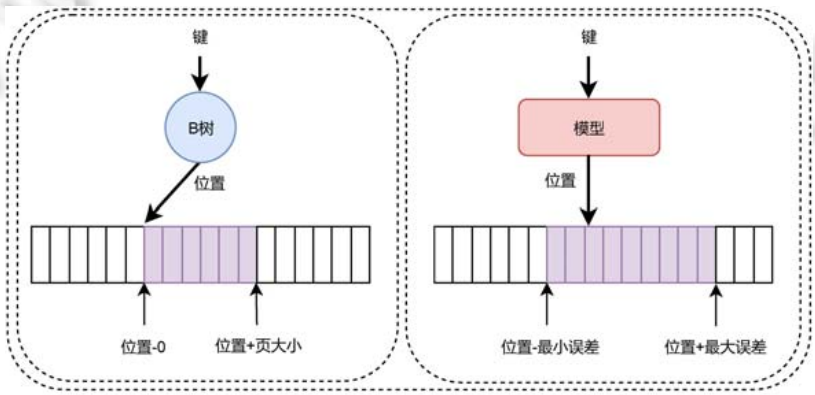


Fig.1 B tree index and learned index

图 1 B 树索引与学习索引

这种新的学习索引结构能够对数据分布情况进行学习,从而生成适配度较高的专用索引结构,能够大大提高数据查询的效率,且在内存开销方面也要优于传统结构.传统的树结构索引查找操作是通过一系列比较,而学习索引则是采用直接计算,因此既减少了数据查询时间,也节约了内存资源.虽然学习索引在某些方面优于传统的索引,但仍存在一些局限,使得它在现实场景中的实用性不是很高.

现有的学习索引主要存在两处不足.

1) 可扩展性较差.没有注意对数据的划分处理,某个数据的插入会引发大量数据的位置变化,这也导致了模型之间的依赖度较高.一旦有某个模型需要重新训练,为了维持模型的准确性,所有的模型都要重新训练;
2) 模型过于复杂.目前,在学习索引领域广为人知的工作是 Kraska 等人提出的递归模型索引(recursive

model index,简称 RMI)结构模型[12],但是该结构采用了递归的模型框架,使得一次查找需要遍历多个模型,增加了查询的消耗.

在本文中,我们创新性地提出了 Dabble 模型,一种基于中间层的可扩展索引模型(a middle layer based scalable learned index scheme).为了有效地学习数据分布,Dabble 首先利用 *K*-Means 算法把数据集进行聚类,从而划分为 *K* 个数据区域,使得每个区域内的数据分布尽可能一致.在模型训练阶段,我们采用前馈神经网络进行学习,并在模型的损失函数中加入数据的访问热度,从而使模型对访问频率高的数据预测更加精准.针对数据插入问题,我们借鉴了 LSM 树[13]中的延迟更新机制,在内存中开辟一块缓存用来存放新插入的数据,当缓存溢出时,一次性将数据进行插入;针对索引更新问题,我们创造性地提出一种基于中间层的机制,通过模型解耦的方式缓解了索引更新带来的消耗.

我们在两个数据集上进行了实验验证,结果表明:

- 在 Lognormal 人工数据集上,Dabble 模型的查询性能比 B+树提升了 69%,比学习索引提升了 13%,比 ALEX[14]提升了 8%;内存占用比 B+树减少了 92%,神经网络模型的内存占用比学习索引减少了 99%,比 ALEX 减少了 98%.同时,Dabble 模型的插入性能比 B+树提升了 250%,比 ALEX 提升了 123%;
- 在真实数据集 Weblogs 上,Dabble 的查询性能比 B+树提升了 49%,比学习索引提升了 33%,比 ALEX 提升了 9%;内存占用比 B+树减少了 92%,神经网络模型的内存占用比学习索引减少了 99%,比 ALEX 减少了 98%.同时,Dabble 的插入性能比 B+树提升了 400%,比 ALEX 提升了 88%.

本文第 1 节将介绍与数据索引相关的一些工作.第 2 节介绍 Dabble 模型的设计.第 3 节将会对索引的更新方式进行讲解.第 4 节通过一系列实验对 Dabble 模型进行评估,并与已有模型进行比较.第 5 节将会对我们的研究内容进行总结,且对未来工作进行规划.

1 相关工作

1.1 B树的改进及变种

Trie[15]使用键的前缀来建树,以减少树结构的内存占用.MassTree[16]和 ART(自适应基树)[17]结合了 B+树和 Trie 的思想来减少缓存缺失.Digital B[18]树使用二进制表示的键,以二进制计算的方式加速 B 树的计算.Graefe[19]则提出用插值搜索替代部分二分搜索,以加快节点内查询.文献[3]提出的混合 B+树索引采用两层 B 树结构:第 1 层将所有记录根据属性分类,第 2 层采用多个 B+树分别处理不同属性类别的记录.

部分学者专注于改进B+树叶节点的数据结构,他们针对不同应用场景对B+树的叶节点做出了适应性的修改,A-树[20]在叶节点中使用线性模型,而 BF-树[21]在叶节点中使用布隆过滤器.还有一部分工作针对 B 树的存储做优化,它们通过优化树索引结构来发挥硬件特性,例如 CPU 缓存、多核处理器、SIMD 和数据预取等.Rao 等人提出了 CSS-树[6],通过使索引节点的大小适应 CPU 缓存大小,并移除节点中的指针改为算术运算来寻找子节点,提高了 B+树在 CPU 缓存上的表现.Rao 等人接着又拓展了静态的 CSS 树,提出了 CSB+树[9]来支持高效的数据更新操作.Chen 等人提出了 pB+树[22],使用更大的索引节点,并通过预取指令在访问节点之前把节点存入缓存中.FAST[5]则利用 SIMD 并行性进一步优化了节点内查找的性能.

1.2 学习索引

RMI[12]学习索引结构利用机器学习模型代替传统的索引结构.如图 2 所示,类似 B 树的分层结构,RMI 建立了一个层次化的模型,下一层的数据划分取决于上一层模型的预测.第 1 层给出一个 0 到 N 的预测,假设第 2 层有 K 个模型,那么第 1 层预测在范围$(0,N/K)$的记录将交给第 2 层的模型 1 进一步预测,落在$(N/K+1,2\times N/K)$的记录将由第 2 层的模型 2 进一步预测,以此类推.类似 B 树,RMI 的每层模型将会进一步缩小搜索范围,每一层被选择的模型接受相同键值的输入,并根据输出选择下一层的模型,直到最后一层给出记录位置的最终预测(通常还会根据最大误差进行二分搜索来确定记录的精确位置).

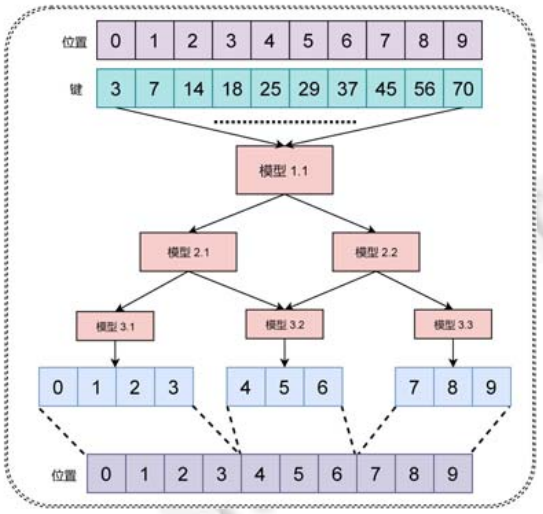


Fig.2 Learned index model

图 2 学习索引模型

虽然 Kraska 等人提出的学习索引模型在查询性能以及内存占用上相较于 B 树取得了很大的提升,但模型仍存在一些问题:它在数据划分上采用均等划分,并没有考虑到数据之间的相关性,不支持数据插入,只能作用于只读负载,因此在使用范围上受到了很大局限.针对索引更新问题,目前一些学者提出了各自的解决方法.

Ding 等人提出的可更新自适应学习索引 ALEX(an updatable adaptive learned index)[14]继承了 RMI 的层级结构,修改了叶子模型(RMI 的最后一层模型)的数据结构,由有序数组变为一个自定义节点结构,并把在误差范围内查找准确位置的搜索方法从二分搜索改成指数搜索.ALEX 模型牺牲了部分空间性能,添加适当的冗余,使它能处理更新操作并提高查询性能.Li 等人提出的自适应独立线性回归模型 AIDEL(adaptive in dependent linear regression models)[23]和 Li 等人提出的自适应单层模型 ASLM(adaptive single layer model)[24]放弃了层次化思路,采用先划分数据再分配给合适的子模型的框架.两者的数据划分算法都采用了贪心算法,ASLM 直接对数据进行贪心算法处理,得到分段的数据;AIDEL 则是将贪心算法用在模型训练阶段,直接得到训练好的模型.

2 模型设计

本节将详细阐述 Dabble 模型的设计框架.与文献[12]一致,本文研究的是单维数据索引,即假设数据集为内存中的一个有序数组.如前所述,我们的目标是尽可能利用数据分布得到一个高效的学习索引模型;同时,模型可以处理数据插入操作,保持一个良好的扩展性.通常,一个数据集的数据分布是非常复杂的,虽然其中蕴含着特定的分布规律,但是仅仅通过一个机器学习模型学习到这个分布规律是非常困难的.其次,不同数据的访问频率是不一样的,如何重点关注热点数据,从而使得学习索引对这些数据的预测精度更高,是提高查询速度的关键之一.对于如何有效地学习数据分布,Dabble 模型中设计了 3 个机制:(1) 利用聚类算法将数据集按照数据的聚集方式划分为 K 个部分,得到 K 个数据区域,使得在每一个数据区域内部,数据分布尽可能相同;(2) 对于每一个数据区域,分别利用神经网络对其进行训练,使网络能够对该数据区域的分布得到一个比较好的拟合;(3) 在神经网络训练阶段,重点关注访问频率高的热点数据,从而使神经网络对这些数据的预测精度更高.另外,为了更快地处理动态插入以及更新操作,Dabble 模型提出了一种基于中间层的数据插入机制,从而使得不同的神经网络模型之间解耦,模型之间保持独立性.当数据插入时,只需要重新训练有数据插入的那个数据区域对应的模型即可,不需要对整个 Dabble 重新训练,从而提高了模型的可扩展性.

图 3 描述了 Dabble 模型的整体框架.

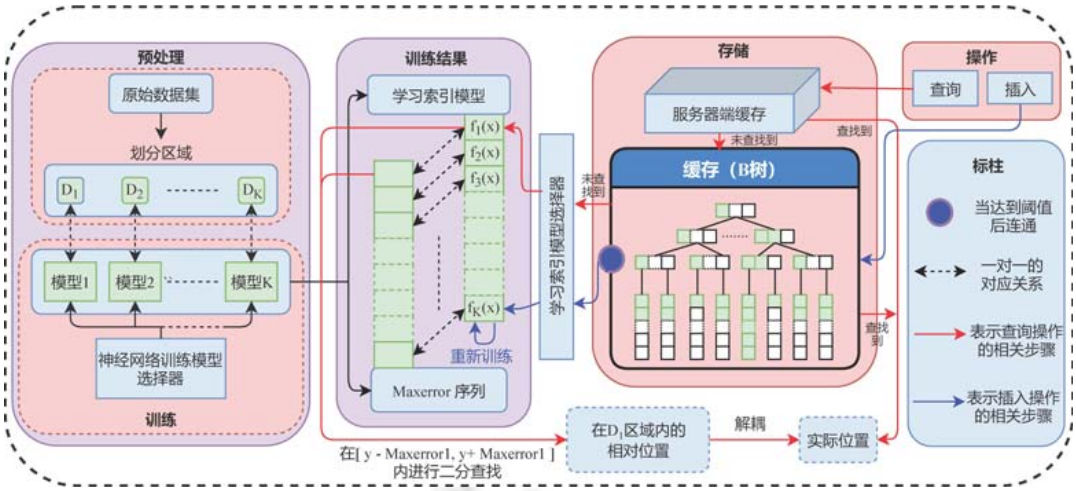


Fig.3 Framework of Dabble

图 3 Dabble 模型框架

从图中可以看出,Dabble 模型主要分为 4 部分:(1) 首先,利用聚类方法对数据集进行预处理,通过 *K*-Means 聚类,将数据集按照其分布规律划分为不相交的区域;(2) 初始化 *K* 个神经网络模型,分别对上一步得到的 *K* 个数据区域进行训练,得到 *K* 个学习索引模型;(3) 当数据查询时,通过模型选择器选出对应的模型,并预测出键对应的记录位置,并通过二分查找在误差范围内找到最终的位置;(4) 插入新的数据时,数据首先放入一个 B 树的缓存中,当缓存达到一定的阈值后,将缓存中的数据一次性归并到数据集中,并对相应的模型进行重新训练.接下来,我们将对模型的每一部分逐一进行介绍.

2.1 数据空间划分

如上所述,我们的目标是尽可能学习到数据的分布,从而使学习索引模型能够在内存占用率小的条件下更快地响应用户请求.但在现实世界中,数据分布往往是比较复杂的,很难通过一个通用的模型较好地学习到数据分布.图 4(a)展示了一个日志记录数据集的键-值分布,其中,键(key)是每一条记录的生成时间,即时间戳(timestamp);值(value)代表了每一条记录的存储位置.通过图中可以看到:键的分布是有区域性的,反映在日志记录数据集上可以理解为对数据集的访问具有时间属性,在不同时间段数据的访问频率是不一样的;其次,从宏观角度看,图 4(a)中的数据分布是比较杂乱的,呈现出一种非线性特点.但是,把图 4(a)中的某一局部信息放大来看,可以发现局部区域数据呈现较好的线性分布,因此最直观的想法是利用简单函数对每一个区域分段学习拟合.综上,如果不对数据进行预处理,直接对整个数据集进行学习,训练出的模型预测精度往往比较低.

通过上面的分析可知,在模型训练之前,我们需要对数据集进行划分.按照数据的分布情况,将整个数据集划分为不相交的子数据区域.在每一个数据区域内部,数据的分布近似一致,而不同的数据区域之间数据分布不一定相同.如图 4(b),通过数据划分算法,将数据集分成了 *K* 个数据区域$\{D_1,D_2,\ldots,D_K\}$,使得每一个子区域内的数据分布尽可能一致.在这种情况下,在每一个数据区域上进行模型训练,使每一个数据区域对应的模型获得较高的预测精度,进而提高查询速度.通过图 4 我们看到,数据分布是有区域性的,比较合理的一种做法是按照数据的聚集分布进行划分.本文采取 *K*-Means 聚类算法,这是一种简单、有效的无监督算法,可以通过调整参数 *K* 调整簇的个数.如学习索引文献[12]中提到的一样,我们假设数据集是内存中的一个排序数组,算法 1 展示了数据集的聚类过程.图 4(b)中展示了经过聚类算法处理后得到的数据划分,可以看到:整个数据集按照分布特点划分成了 *K* 个区域$\{D_1,D_2,\ldots,D_K\}$,并且数据区域间不相交且大小有序,即第 i+1 个数据区域中的数据比第 i 个数据区域中的数据都要大.

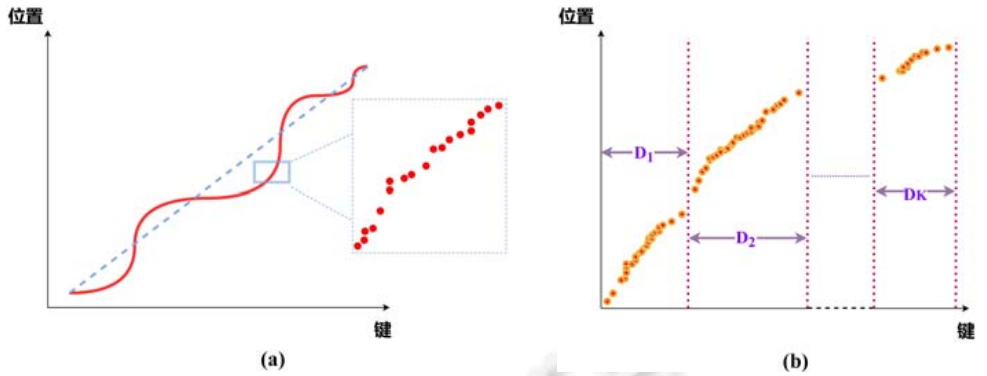


Fig.4 Data space partitioning

图 4 数据空间划分

下面展示了算法的具体过程.假设训练数据集是$\{x^{(1)},x^{(2)},\ldots,x^{(n)}\}$,$x^{(i)}$代表了第 i 个数据记录,$c^{(i)}$代表了训练数据 i 与 K 个类中最近的那个类,质心 u_j 代表了每一个簇中样本的中心点.在每一轮迭代中,对质心进行更新直至算法收敛.直观上理解,K 值越大,会使神经网络模型对应的数据区域越小,数据分布越一致,模型拟合效果越好.但是 K 值过大会使模型的数目过多,从而使模型选择器在选择某个数据查询对应的模型时花费时间过多.不同数据集的数据分布是不同的,针对不同的数据集,K 值需要进行调整,我们将在后面的实验部分展示这一过程.另外,在数据集特别大的情况下,可以采取 K-Means 的优化算法,如 Mini Batch K-Means 等加速聚类的过程.注意:这里对数据集进行划分只是一个虚拟概念上的划分,整个数据集依旧是一个整体的排序数组.

算法 1. 数据集聚类过程.

初始化:随机选取 K 个数据点作为聚类之心$\{u_1,u_2,\ldots,u_K\}$.

输出:K 个数据区域.

1. 重复以下过程,直至模型训练收敛 {
2. 对于每一个数据 i,计算其属于的类 $c^{(i)}$:
3. $c^{(i)}=\arg\min_j \| x^{(i)}-u_j \|^2$;
4. 对于每一个类 j,重新计算该类的质心:
5. $u_j=\dfrac{\sum_{i=1}^{K}1\{c^{(i)}=j\}x^{(i)}}{\sum_{i=1}^{K}1\{c^{(i)}=j\}}$;
6. }

2.2 模型训练

本节详细阐述 Dabble 模型的训练过程.经过上一节的聚类过程,整个数据集按照数据分布被划分为 K 个不相交的数据区域,每一个区域内的数据分布近似一致.接下来,对于每一个区域,我们使用一个神经网络模型对其进行学习.其中,模型的输入是数据记录的键,对应的训练标签是该键的位置(position),如图 5 所示,模型的输入与输出均为一维的标量类型.其中,神经网络模型采用两层架构,即:在输入层与输出层中间只加入一层隐藏层的神经元,并且在隐藏层后面加一层 ReLU 激活函数用来做非线性变换.可以看到,用来训练的神经网络模型是较为简单的,原因如下:(1) 如文献[23,24]所述,当把数据划分以后,每一个区域的分布可以使用简单的模型就可以学习到,但是由于数据的局部区域分布依然可能存在非线性的特征,因此需要引入激活函数来学习非线性特征;(2) 众所周知,神经网络模型的训练是非常消耗时间的,而采用简单的模型可以加速网络的训练;3) 在数据查询阶段,简单的模型可以提高计算速度,有效缩短查询时间.

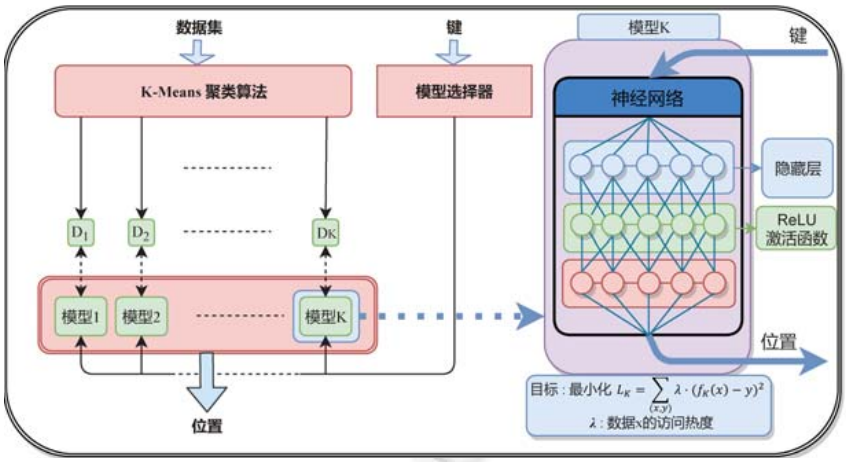


Fig.5 Model training

图 5 模型训练

我们的目标是使模型预测出来的位置与数据记录对应的位置尽可能接近,即在训练过程中,优化的目标是最小化 $L_i = \sum_{(x,y)} (f_i(x) - y)^2$,其中,x,y 分别代表训练数据的数据键以及对应的存储的位置,$f_i(x)$代表第 i 个训练模型(i 取值为 1 到 K),L_i 为第 i 个模型的损失函数.即:训练的结果是使每一个数据区域对应的神经网络模型对该区域的预测精度尽可能高.但这里存在一个问题:在大型数据库应用系统中,客户端对数据的访问请求往往呈现某种规律,如齐夫定律(Zipf's law).在这种情况下,某些数据会成为热点数据,而有些数据很少被访问到.受到这规律的启发,我们创新性地将数据的访问模式加入到模型的训练过程中,如图 5 所示,当前的最小化目标变为 $L_i = \sum_{(x,y)} \lambda_x \cdot (f_i(x) - y)^2$,其中,λ_x 代表了单个数据键 x 的访问热度,每一个数据键 x 对应的λ_x 值不同,该值可以通过对 x 的历史访问数据计算得出.加入这个训练因素的好处如下:(1) 如上所述,数据的访问热度蕴含了数据请求的访问模式信息,可以使模型更好地适应数据分布;(2) 模型更加关注于热点数据,那么对于热点数据的预测精度相较于普通数据会更高.在数据查询阶段,由于对热点数据访问速度更快,平摊下来会提高整个系统的请求响应效率.

当 K 个神经网络模型 $f_i(x)$,1<=i<=K 训练完成后,每一个模型都会对应一个误差值(MaxError),这个值代表了模型 $f_i(x)$在第 i 个数据区域最大的预测误差.在下一节中,我们将通过 $f_i(x)$,1<=i<=K 和对应的误差值完成对数据查询的操作.

2.3 数据查询

通过上一节,我们得到了 K 个索引模型 $f_i(x)$,1<=i<=K,以及每一个模型对应的误差值.通过第 2.1 节得知,经过聚类得到的 K 个数据区域之间大小有序,并且每一个区域都有对应的键值的范围(MinKey, MaxKey).如图 5 所示,当一个查询到来时,我们首先通过模型选择器比较判断键的大小,从而找到该查询对应的索引模型 $f_i(x)$,然后利用 $f_i(x)$计算得到该数据记录所对应的存储位置 y.注意,此时得到的位置是一个近似的位置,我们接下来需要通过查找算法例如二分查找在[y−MaxError,y+MaxError]范围内查找数据的最终位置,从而完成查询操作.

另外,为了提高查询效率,Dabble 模型会在服务器端缓存(cache)近期访问的数据以及热点数据.在这种情况下,每当查询到来的时候,首先查询系统缓存:如果数据命中,则直接返回结果;否则,访问 Dabble 索引进行数据的查询.为了防止缓存与真实数据的不一致性,我们采用了租赁机制[25].在租赁有效期间,缓存数据是有效的,并且该缓存对应的数据更新是被锁定的;租赁失效后,如果想要重新缓存该部分数据,需要重新向系统发出租赁申请.在这种机制下,我们可以保证用户访问到的数据是最新的.

3 索引更新

通过机器学习的方式构建索引,可以有效地学习到数据分布,从而在内存占用很小的前提下提供高效的查询速度.但这里面存在一个问题:新数据的不断插入,会引起数据分布的改变.在这种情况下,之前训练的神经网络模型将出现预测位置偏移的情况,并且会随着数据分布改变的程度不断加大.因此,朴素的想法是在新数据集上重新训练模型,但这个耗费是非常大的.文献[12]提出的模型假设数据库是只读型的,对于索引更新的情况并没有给出很好的解决方案;文献[14]采用预留位置的方法用来存储新插入的数据,但是当某个区域插入数据过多的时候会出现溢出问题,而且该方法也会造成内存的浪费.可以看出,对于学习索性模型,索引更新问题是一个棘手的问题.在本文中,我们提出一种基于中间层的方法来尽可能缓解索引更新带来的影响.

3.1 模型解耦

在第 2.3 节中我们提到:当新查询到来的时候,经过模型选择器选择模型 $f_i(x)$,计算得到键 x 对应的位置,并在误差范围内进行查找,从而获得最终的数据.在数据预处理阶段,我们通过聚类方法把数据集分割成 K 个区域,那么能不能在数据插入的时候,只更新当前的模型 $f_i(x)$?这样,其余 K–1 个模型可以保持不变,从而将更新带来的消耗尽可能降低.

受到程序设计领域模型解耦思想的启发,我们可以在模型的预测部分加一层中间层,使模型之间相互独立,这样每个模型预测出来的位置变为这个键在该模型对应的数据区域的位置.在这种情况下,如图 5 所示,模型 $f_i(x)$的输出值 y 代表的是键 x 在第 i 个数据区域 D_i 中的位置.令$|D_i|$代表第 i 个数据区域的大小,那么键 x 真实的预测位置可以通过如下计算得到: $pos=\sum_{t=1}^{i-1}|D_t|+y$,其中,$\sum_{t=1}^{i-1}|D_t|$代表了前 i–1 个数据区域的包含的数据量,该累计值再加上 y,得到键 x 真正的预测存储位置.通过这种方式,我们只需在聚类以及后续的索引更新过程中记录下来每个数据区域的大小,即可方便地通过中间层得到查询的预测位置.可以看到:通过加了一层中间层,模型 $f_i(x)$由预测键 x 在整个数据集中的位置变为预测在区域$|D_i|$中的相对位置.在这种情况下,新插入的数据只会影响数据区域$|D_i|$中的数据分布,其他模型依然可以通过预测出来的数据键的相对位置加上位于前面的所有数据区域的大小得到数据键的真正位置.模型之间互不影响,相互独立,为后续的数据插入过程提供了高效的解决方法.另外,这里提到的中间层只是一个虚拟上的概念,并没有真正改变数据的存储位置,因此不会影响数据的分布.与 ASLM[24]、AIDEL[23]不同,Dabble 模型最终输出的 pos 代表在整个数据集的真实位置.因此,Dabble 模型既适用于存储在连续内存中的数据集,也可以应用于将数据区域分块存储的场景.

3.2 数据更新

通过模型解耦,模型 $f_i(x)$,$1\leqslant i\leqslant K$ 之间相互独立,当插入数据的时候,我们只需要重新训练对应的神经网络模型即可,而不需要在全部数据集上进行训练.但是,如果插入操作比较多的时候,对模型频繁地重新训练将会带来非常大的代价,严重影响系统的响应速度.基于日志结构的合并树 LSM Tree(log structured merge trees)[13]是一种写优化的数据组织方式,被应用于多种数据库,如 LevelDB,Hbase,Cassandra 等.LSM Tree 中的一个设计思想是延迟更新,即在内存中开辟一块缓存,用来存储新插入的数据.在这种情况下,写操作可以高效地进行,当缓存达到阈值后,一次性将他们归并到磁盘中.

在本文中,我们借鉴了 LSM Tree 的思想.首先,在内存中开辟一块缓存(buffer),并将其分为 K 个部分,作为 K 个模型的缓冲区.当有新数据插入的时候,首先通过模型选择器获得新数据应该存储的数据区域,然后直接将该条数据放入到对应的缓存块中.当某个缓存块达到阈值的时候,将其中的数据归并到真正的数据区域中,并对相应的模型 $f_i(x)$重新训练,使其能够拟合当前的数据分布.在这种情况下,当某个数据区域由于数据分布过于复杂,导致一个神经网络模型无法较好地学习时,对该数据区域进行分裂操作,此时,整个数据集合变为 K+1 个区域.注意:如第 2.1 节提到的一样,把数据集划分为区域,只是一个逻辑上的行为,数据集依旧是作为一个整体存储.

引入数据更新机制后,对于数据查询请求,需要首先在第 2.3 节中提到的系统缓存进行查找,如果失败,则在数据缓存块中进行查找:如果查找成功,则直接返回结果;否则,使用 Dabble 模型进行查找.为了提高在缓存中的

查询速度,缓存中的数据可以按照 B 树方式进行组织.

4 实验评估

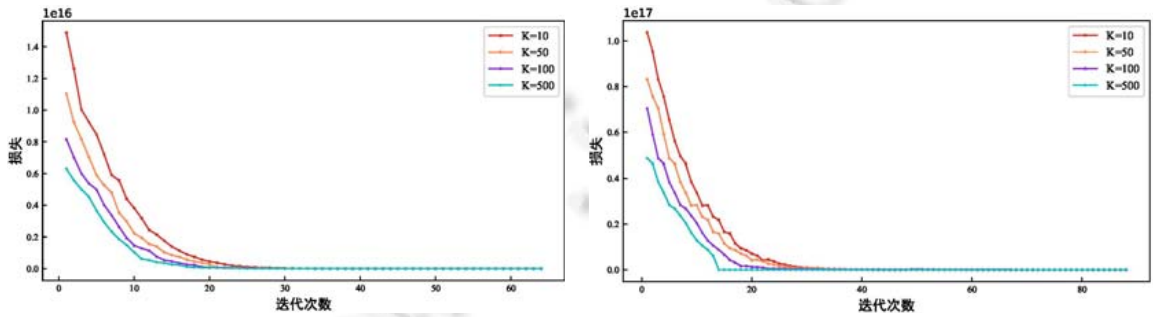
在本节,我们通过一系列实验评估 Dabble 模型的性能,并与 B+树、学习索引在读、写性能上进行对比,从而验证模型的有效性.实验在一台 Intel Xeon,CPU E5-2620 v4@2.10GHz,64G,32 Cores Dell 服务器上进行.其中,Dabble,学习索引和 ALEX 的神经网络模型使用 Pytorch 1.0 编程实现,并在一块 8G GTX 1080 GPU 上进行训练.为了保证实验的公平性及有效性,3 个模型的性能测试均使用 Python 3.6 语编程实现,并使用单线程在 CPU 上进行模拟测试.3 个模型的具体设置如下.

1) Dabble:每一个模型为两层神经网络,输入与输出为对应键和值,中间隐藏层大小为 32 个神经元,后面连接一个 ReLU 层用来做非线性变换,输出层代表数据键对应的位置.学习率为 0.0001,批训练大小为 1 000,优化函数使用 *Adam*.未设置缓存;
2) 学习索引:使用两层模型的递归结构,第 1 层为一个带双隐层的神经网络,中间隐藏层大小为 16×16 个神经元,每个隐层后使用 ReLU 作为激活函数;第 2 层为 100 个不含隐层的线性模型,第 2 层的输出代表数据键对应的位置.未设置缓存;
3) B+树:每个节点包含的记录数(page size)为 256,对 B+树进行了缓存优化,缓存大小设置为 64K 字节;
4) ALEX:基于学习索引模型实现,将学习索引模型中采用的 RMI 模型[8]替换成 Adaptive RMI[14]模型,且叶节点采用规模为键值数目 4 倍的 GA(gapped array)作为存储结构.未设置缓存.

我们采用不同的数据集进行测实验证,分别为 Lognormal 人工数据集和真实数据集 Weblogs.其中:Lognormal 数据集按照对数正态分布(均值为 0,方差为 2)的方式采样了 500 万条不重复的数据;Weblogs 数据集使用了 Live Maps Back End 在一天 24 小时的访问日志记录,大小为 304M,并提取里面的时间戳作为键,记录数为 102 万.实验按照以下方式进行:首先,分别对 Dabble 模型、学习索引模型和 ALEX 模型在两个数据集上进行训练,并保留训练好的模型,同时在两个数据集上建立 B+树;然后,生成不同大小的读、写工作负载对模型进行性能测试;并调整 *K* 值测试 Dabble 模型在不同数据划分下的性能表现.

4.1 Dabble模型训练

首先,我们展示了 Dabble 模型在两个数据集上的训练过程,如图 6 所示.其中,横轴代表了训练的轮数,纵轴代表了模型的损失值.我们分别设置了 *K* 为 10,50,100,500.



(a) Lognormal 数据集 (b) Weblogs 数据集

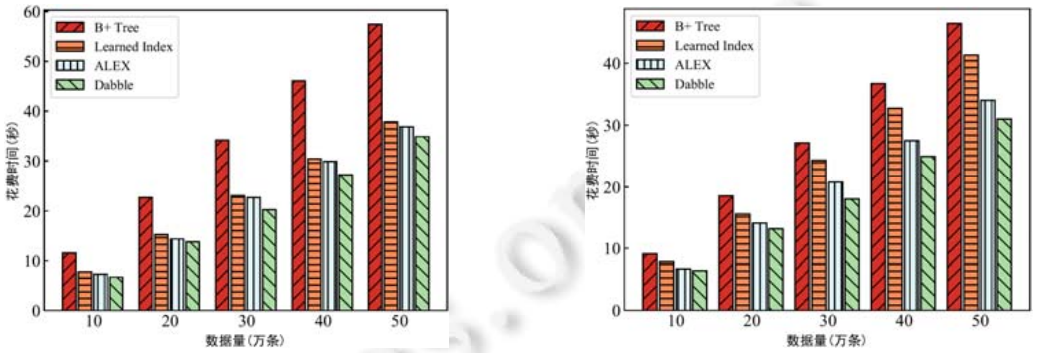
Fig.6 Model training process

图 6 模型训练过程

从图中可以看到:随着训练过程的进行,模型最终收敛到一个固定的损失值,较好地拟合了数据的分布.训练过程与 *K* 值的大小有关:*K* 值越大,模型收敛的速度越快.原因在于:*K* 值越大,每一个数据区域所包含的数据记录越少,因此模型在小的数据规模上学习速度更快.另外,如图 6(a)和图 6(b)所示,因为 Weblogs 数据集相对较小,模型在 Weblogs 数据集上收敛速度比在 Lognormal 数据集上更快.

4.2 查询性能测试

本节实验测试了 Dabble,学习索引,B+树和 ALEX 的查询性能.如图 7 所示,横轴代表查询记录的数目,纵轴代表完成所有查询需要的时间.时间越小,代表吞吐量越大.实验中,我们分别在两个数据集上生成了 10 万~ 50 万条查询负载,其中,Dabble 模型的 K 值在 Lognormal 数据集上设置为 100,在 Weblogs 数据集上设置为 50.



(a) Lognormal 数据集 (b) Weblogs 数据集

Fig.7 Performance comparison of data query

图 7 数据查询性能对比

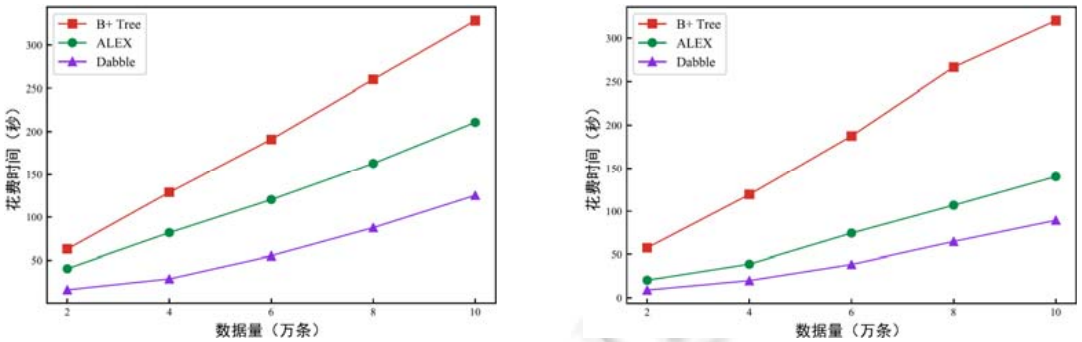
从图中可以看到,Dabble 模型、学习索引模型以及 ALEX 在查询性能上均优于 B+树.原因在于:学习索引模型能够较好地学习到数据集潜在的数据分布规律,从而训练出适应于当前数据集的神经网络模型.当查询的时候,能够以更小的时间复杂度找到记录的位置,而 B+树的查询性能固定为 $Log(N)$.在 Lognormal 数据集上,Dabble 模型比 B+树提升了 69%,比学习索引模型提升了 13%,比 ALEX 提升了 8%;在 Weblogs 数据集上,比 B+树提升了 49%,比学习索引提升了 33%,比 ALEX 提升了 9%.另外,随着查询负载的增大,学习索引模型相比于 B+树的优势越来越明显.原因在于:学习索引模型一旦被训练出来,其查询的过程本质上是在做矩阵运算(神经网络预测的过程),因此其查询时间与查询负载保持一个近似线性的关系增长.

从图 7(a)和图 7(b)中可以看到,Dabble 模型在 Lognormal 和 Weblogs 两个数据集上性能均优于学习索引模型以及 ALEX 模型.原因如下:(1) 如上文所述,除了使用神经网络学习分布,Dabble 模型在数据预处理阶段使用聚类算法划分数据集为 K 个数据区域,从而使神经网络对每个数据区域更好地学习到分布;(2) 在训练阶段,我们创造性地将数据记录的访问热点融入到模型中,从而使模型对访问频率大的数据预测精度更高,从而加速查询;(3) 相比于学习索引和 ALEX 的多层递归模型,Dabble 模型使用了两层的架构,并且只在第 2 层使用神经网络进行预测过程,因此在查询的过程中更加便捷快速.ALEX 模型采用 Gapped Array 的方式在叶子节点组织数据,能更好地的适应数据分布,因此在查询性能方面优于学习索引模型.另外,从图 7(a)和图 7(b)中可以看出:Leaned Index 在 Weblogs 数据集上表现的更差一些,其性能相较于 Dabble 相差较大.因为 Weblogs 属于真实世界的数据集,因此其数据的分布相对于 Lognormal 分布更加复杂多变.在这种情况下,Dabble 模型由于上面所述的机制的保证,能够更好地学习到数据的分布规律,从而获得更好的查询性能.

4.3 数据插入性能测试

本节实验对比模型在数据插入时的性能.因为学习索引不能处理数据插入问题,因此我们比较了 Dabble、B+树和 ALEX 的数据插入性能.如图 8 所示,横轴代表了数据的插入数目,纵轴表示完成全部数据插入所需要的时间.我们分别在两个数据集上生成了 2 万~10 万条插入记录,其中,Dabble 模型的 K 值在 Lognormal 数据集上设置为 100,在 Weblogs 数据集上设置为 50.在第 3.2 节中我们提到:在数据插入过程中,如果 Dabble 的缓存块达到阈值,会把缓存中的数据归并到真正的数据集中重新训练模型.因此为了保证实验对比的公平性,我们把模型重新训练的时间也加以记录.其中,在 Lognormal 数据集上神经网络模型训练一次的时间为 15.68s,在 Weblogs

数据集上训练一次的时间为 8.77s.



(a) Lognormal 数据集 (b) Weblogs 数据集

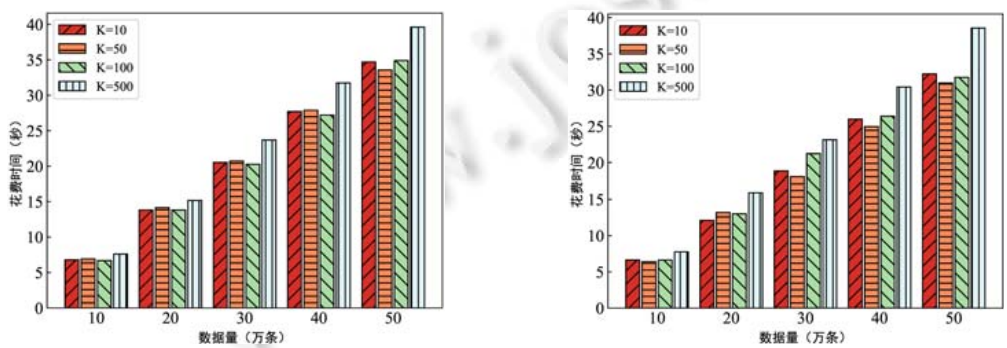
Fig.8 Performance comparison of data insertion

图 8 数据插入性能对比

从图 8(a)和图 8(b)中可以看到:在数据插入性能上,Dabble 优于 B+树很多.其中:在 Lognormal 数据集上,相比于 B+树提升了 250%,比 ALEX 提升了 123%;在 Weblogs 数据集上,相比于 B+树提升了 400%,比 ALEX 提升了 88%.原因在于,Dabble 模型借鉴了 LSM 树中的延迟更新机制.如第 3.2 节所述,我们在内存中开辟了一块区域作为数据插入的缓存,并将其分为 K 个缓存块,每一个缓存块用 B 树进行组织.当有数据插入的时候,首先根据模型选择器,得到数据需要插入的区域,然后将它直接插入到对应的缓存块中.当某个缓存块存储达到阈值的时候,我们一次性将其归并到真正的数据集中,并对其对应的神经网络模型进行重新训练.在这种情况下,数据的插入不会频繁引起模型的重训练,从而加速了插入的速度.而 B+树为了维护索引树的平衡,需要在数据插入时同步进行树结构的调整,从而在数据插入时,性能相比于 Dabble 要差很大.ALEX 虽然实现了模型的插入功能,但其本质上是一个递归的 RMI 模型,因此插入性能要劣于 Dabble.另外,两个模型在 Lognormal 数据集上的性能要差于 Weblogs 数据集,因为 Lognormal 数据集更大,因此在索引更新时,需要花费更大的代价进行维护.

4.4 不同K值对Dabble的影响

本节实验验证在不同 K 值下,即聚类数目不同时对 Dabble 性能的影响.与第 4.2 节一致,我们分别在两个数据集上测试 Dabble 在不同的 K 值下数据查询的性能,查询负载为 10 万~50 万条,实验结果如图 9 所示,我们分别设置了 K 为 10,50,100,500.



(a) Lognormal 数据集 (b) Weblogs 数据集

Fig.9 Analysis of K for Dabble

图 9 K 值不同对 Dabble 的影响

从图中可以看到:随着查询负载的增加,查询时间近似成线性增加.原因如第 4.2 节中所述,查询的消耗主要由神经网络模型中的矩阵运算决定,而该过程与查询请求的数量成一个正比的关系.如图 9(a)所示,在

Lognormal 数据集中,模型在 K 设置为 100 的时候近似达到最优的效果.原因如下:K 值决定了数据集划分的个数,显然,K 值越大,数据集划分的越细,从而单个神经网络模型在对应的数据区域上学习效果越好,预测越精准.但是从图 9(a)中可以看出:模型的性能在 K 值从 10 增长到 100 时是增加的,但是从 100 变为 500 时反而变差.原因在于:虽然 K 值越大使神经网络学习效果越好,但是当 K 过大时,会由于模型数目过多,使得模型选择器在选择某个查询对应的模型时花费的时间增加.因此,对于某个特定数据集,存在某个 K 值使模型性能达到最优.另外,从图 9(b)可以看到:在 Weblogs 数据集上,Dabble 模型在 K 设置为 50 的时候性能近似最优.原因在于两个数据集的数据分布不同,并且数据规模大小不一样,因此在不同的数据集上,最优性能对应的 K 值是不同的.

4.5 模型内存占用对比

本节对比了 4 个模型内存占用的情况,结果见表 1.其中,24M 字节与 5.3M 字节分别代表了两个数据集在模型中的内存占用,4K 字节、20K 字节、301K 字节代表了分别代表不同模型的神经网络的内存占用.

Table 1 Comparison of memory overload

表 1 内存占用对比

模型		内存占用	
		Lognormal	Weblogs
Dabble	K=10	24M+4K	5.3M+4K
	K=50	24M+4K	5.3M+4K
	K=100	24M+4K	5.3M+4K
	K=500	24M+20K	5.3M+20K
学习索引		24M+301K	5.3M+301K
ALEX		29M+207K	10.3M+207K
B+树		307M	70.7M

从表中可以看到,B+树的内存占用相对于学习索引结构要高出很多.原因在于:B+树中除了存储真正的数据键外,还需要存储指针等一系列附加信息.而且数据集规模越大,B+树的节点越多,从而占据大量的内存.而学习索引结构由于只存储神经网络模型参数和数据键,因此内存占用情况得到了缓解.另外,如上面所述, Dabble 模型只设置了两层的架构,并且只在第 2 次使用两层神经网络,因此比学习索引内存占用更小. ALEX 模型由于采用动态的 RMI 递归层次模型,因此神经网络模型占用要比学习索引小一些,但是 ALEX 需要在叶子节点为新插入的数据预留存储位置,因此数据的内存占用相较于学习索引模型要更大.

4.6 小 结

在本节,我们通过一系列实验对比了 Dabble,学习索引以及 B+树在数据查询以及数据插入方面的性能.实验结果表明:

- 在 Lognormal 人工数据集上,Dabble 模型的查询性能比 B+树提升了 69%,比学习索引提升了 13%,比 ALEX 提升了 8%;内存占用比 B+树减少了 92%,神经网络模型的内存占用比学习索引减少了 99%,比 ALEX 减少了 98%;同时,Dabble 模型的插入性能比 B+树提升了 250%,比 ALEX 提升了 123%;
- 在真实数据集 Weblogs 上,Dabble 的查询性能比 B+树提升了 49%,比学习索引提升了 33%,比 ALEX 提升了 9%;内存占用比 B+树减少了 92%,神经网络模型的内存占用比学习索引减少了 99%,比 ALEX 减少了 98%;同时,Dabble 的插入性能比 B+树提升了 400%,比 ALEX 提升了 88%.

另外,我们还对比了不同聚类数目对模型性能的影响,但是如何准确设置 K 值从而获取最优性能,仍是一个有待解决的问题.

5 总结与未来工作

如何设计一种高效的数据索引结构,一直以来是数据库领域的研究热点.Kraska 等人首次提出利用神经网络训练学习索引模型,从而代替 B 树等传统索引结构,取得了引人注目的效果.但是,当索引更新导致数据分布改变的时候,其模型需要重新训练神经网络以适应新的数据分布,因此消耗巨大.本文提出一种基于中间层的可扩

展的学习索引模型 Dabble.为了更好地学习数据的分布,我们提出了 3 种机制:首先,利用 *K*-Means 聚类算法,根据数据分布将数据集划分为 *K* 个区域;其次,利用神经网络模型分别学习 *K* 个区域的数据分布;最后,在训练阶段把数据的访问热点程度融入到神经网络中,使其对访问越频繁的数据预测精度越高.在索引更新阶段,我们提出一种基于中间层的机制,即,通过使神经网络模型输出预测的相对位置实现模型之间的独立.从而,当某个数据区域由于数据插入引起数据分布改变的时候,只需要重新训练当前的模型即可.实验结果表明,与其他方法相比,Dabble 模型在内存占用、数据查询以及插入等情况下都取得了显著的提升.

如上所述,*K* 值是影响 Dabble 模型性能的一个关键参数.*K* 值的大小决定了聚类时划分数据区域的数目以及后续的模型训练过程:*K* 值设置的过小或者过大,都会降低模型的性能.对于不同的数据集,存在一个特定的 *K* 值,使模型在该数据集上获得最优的性能.目前,*K* 值的设定是通过在实验中进行调整,从而获取一个近似最优的值.未来,我们将考虑从数学的角度,尝试推导得出理论上最优的 *K* 值,从而能够在实验过程中更快地寻找到最优 *K* 值,使模型获得最佳性能.

References:

[1] Graefe G, Larson PA. B-Tree indexes and CPU caches. In: Proc. of the Int'l Conf. on Data Engineering (ICDE). IEEE, 2001. 349−358.

[2] Zhou D, Liang ZC, Meng XF. HF-Tree: An update-effcient index for flash memory. Journal of Computer Research and Development, 2010,47(5):832−840 (in Chinese with English abstract).

[3] Changsun NN, Zhang YK, Hua DX. Hybrid index structure based on B+ tree. Computer Engineering, 2012,38(14):35−40 (in Chinese with English abstract).

[4] Chang YC, Chang YW, Wu GM, Wu SW. B*-Trees: A new representation for non-slicing floorplans. In: Proc. of the Annual Design Automation Conf. (DAC). ACM, 2000. 458−463.

[5] Kim CY, Chhugani J, Satish N, Sedlar E, Nguyen AD, Kaldewey T, Lee VW, Brandt SA, Dubey P. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In: Proc. of the Conf. on Management of Data (SIGMOD). ACM, 2010. 339−350.

[6] Rao J, Ross KA. Cache conscious indexing for decision-support in main memory. In: Proc. of the Int'l Conf. on Very Large Data (VLDB). Morgan Kaufmann/ACM, 1999. 78−89.

[7] Alexiou K, Kossmann D, Larson PÅ. Adaptive range filters for cold data: Avoiding trips to Siberia. Proc. of the VLDB Endowment (PVLDB), 2013,6(14):1714−1725.

[8] Fan B, Andersen DG, Kaminsky M, Mitzenmacher M. Cuckoo filter: Practically better than bloom. In: Proc. of the Int'l Conf. on Emerging Networking Experiments and Technologies (CoNEXT). ACM, 2014. 75−78.

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

[10] Zhang HC, Andersen DG, Pavlo A, Kaminsky M, Ma L, Shen R. Reducing the storage overhead of main-memory OLTP databases with hybrid indexes. In: Proc. of the Conf. on Management of Data (SIGMOD). ACM, 2016. 1567−1581.

[11] Wu XB, Ni F, Jiang S. Wormhole: A fast ordered index for in-memory data management. In: Proc. of the European Conf. on Computer Systems (EuroSys). ACM, 2019. 1−16.

[12] Kraska T, Beutel A, Chi EH, Dean J, Polyzotis N. The case for learned index structures. In: Proc. of the Conf. on Management of Data (SIGMOD). ACM, 2018. 489−504.

[13] Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE. Bigtable: A distributed storage system for structured data. ACM Trans. on Computer Systems (TOCS), 2008,26(4):1−26.

[14] Ding JL, Minhas UF, Zhang HT, Li YN, Wang C, Chandramouli B, Gehrke J, Kossmann D, Lomet DB. ALEX: An updatable adaptive learned index. arXiv preprint arXiv:1905.08898, 2019.

[15] Knuth DE. The Art of Computer Programming: Sorting and Searching. Vol.3, Pearson Education, 1997.

[16] Mao YD, Kohler E, Morris RT. Cache craftiness for fast multicore key-value storage. In: Proc. of the European Conf. on Computer Systems (EuroSys). ACM, 2102. 183−196.

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

[18] Lomet DB. Digital B-trees. In: Proc. of the Int'l Conf. on Very Large Data (VLDB). Morgan Kaufmann Publishers/ACM, 1981. 333−344.

[19] Graefe G. B-Tree indexes, interpolation search, and skew. In: Proc. of the Int'l Workshop on Data Management on New Hardware (DaMoN). ACM, 2006.

[20] Galakatos A, Markovitch M, Binnig C, Fonseca R, Kraska T. A-Tree: A bounded approximate index structure. arXiv preprint arXiv: 1801.10207, 2018.

[21] Athanassoulis M, Ailamaki A. BF-Tree: Approximate tree indexing. Proc. of the VLDB Endowment, 2014,7(14):1881−1892.

[22] Chen SM, Gibbons PB, Mowry TC. Improving index performance through prefetching. In: Proc. of the Conf. on Management of Data (SIGMOD). ACM, 2001. 235−246.

[23] Li PF, Hua Y, Zuo PF, Jia JN. A scalable learned index scheme in storage systems. arXiv preprint arXiv:1905.06256, 2018.

[24] Li X, Li JD, Wang XL. ASLM: Adaptive single layer model for learned index. In: Proc. of the Database Systems for Advanced Applications (DASFAA). Springer-Verlag, 2019. 80−95.

[25] Ren K, Zheng Q, Patil S, Gibson G. IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion. In: Proc. of the Int'l Conf. for High Performance Computing, Networking, Storage and Analysis (HPCC). IEEE, 2014. 237−248.

附中文参考文献:

[2] 周大,梁智超,孟小峰.HF-Tree:一种闪存数据库的高更新性能索引结构.计算机研究与发展,2010,47(5):832−840.

[3] 长孙妮妮,张毅坤,华灯鑫.一种基于 B+树的混合索引结构.计算机工程,2012,38(14):35−40.

高远宁(1994－),男,山东平阴人,博士生,主要研究领域为数据工程,数据库索引.

叶金标(1998－),男,本科生,主要研究领域为数据库索引,机器学习.

杨念祖(1999－),男,本科生,主要研究领域为数据库索引,机器学习.

高晓沨(1982－),女,博士,教授,博士生导师,CCF 专业会员,主要研究领为数据库索引,算法设计与优化.

陈贵海(1963－),男,博士,教授,博士生导师,CCF 会士,主要研究领为云计算,分布式系统,无线网络.