East China Normal University
Database System Implementation, Spring 2020

学习式索引结构：模型、评测与系统

李鹏，10175501102
10175501102@stu.ecnu.edu.cn

2020 年 7 月

摘要

本文重点介绍 MIT 数据系统组近几年在将机器学习融入索引结构方向上做的一系列工作。文章首先介绍 MIT 数据系统组提出的学习式 B-Tree 索引、Hash 索引、Bloom 过滤器和多维索引，并介绍该组对其做出的一些改进；然后介绍该组提出的面向学习化索引结构的基准评测——SOSD；最后介绍该组开发的一种新型的融入机器学习模型的数据处理系统——SageDB。

关键词：索引结构，机器学习，存储管理，数据库

目录

1 简介

1.1 调研对象

近年来机器学习的发展为拥有 50 多年历史的数据库系统提供了新的发展思路。虽然传统的数据库系统已经基本成熟并且被商业广泛应用，但是随着大数据时代的到来，数据库系统在两个方面存在着新的挑战 [1]：(1) 数据量持续增大期望单个查询任务具有更快的处理速度，而传统的数据库系统不能有效利用现代的硬件加速器；(2) 数据分布和查询负载的迅速变化及其多样性使得基于 DBA 经验的数据库配置和查询优化偏好不能实时地调整为最佳运行时状态。而当下的机器学习技术恰好同时能够解决这两个问题：应用现代加速器以及从众多参数调节经验中学习，因此近年来越来越多的研究人员开始考虑如何将机器学习技术融入传统的数据库当中 [2]。总结来看，目前将机器学习融入数据库系统的研究工作主要包含三个方面 [1]：存储管理、查询优化的及其学习化研究和自动化的数据库管理系统。本文重点关注机器学习化的存储管理中对索引结构进行的改进。

1.2 研究难点

传统的数据库系统中，有效的数据访问离不开索引结构，数据库系统中存在着多种索引结构来满足不同的访问模式需求 [3, 4]：B 树索引适用于范围搜索，Hash 索引适合点查询，Bloom 过滤器适用于判断某个记录是否存在，而多键索引、kd 树等适合为多维数据建立索引。在过去的几十年中，索引被广泛优化，但大多集中在提高内存、缓存或者 CPU 执行效率，而且目前对索引的优化已经进入瓶颈期，优化空间缩窄。但是现在的索引结构仍然存在两方面问题，一是不能充分利用现代加速器的高并行处理能力，二是传统的索引结构不对数据的分布做任何假设，而目前很多数据库系统中的数据通常具有常见的模式，如何利用数据的分布来对索引进行优化是一个需要考虑的方向。考虑到机器学习模型能够自动挖掘数据的模式，并且能够进一步借助硬件加速器来提高数据库系统的处理速度，因此我们可以考虑将机器学习模型的优势融入到数据库系统的索引结构当中。当然，将机器学习融入索引结构不能够生搬硬套，索引结构不同于机器学习模型而有其自身的特点，比如索引结构要求确定性而机器学习主要是解决非确定性问题，如何解决机器学习和索引结构的适配性问题为数据库研究者带来了新的挑战。本文希望在调研 MIT 数据系统组在这个方向上做的工作来得到一点想法和启发。

1.3 本文工作

在调研相关研究的基础上，本文重点关注了麻省理工学院数据系统组（Data Systems Group@MIT[1]）近年来在将机器学习模型融入索引结构方面做的一些工作：2018 年 Kraska 等人提出的学习式索引结构（learned index structures）[5] 是将机器学习融入数据库系统的开山之作，该工作探讨了如何用机器学习模型对传统数据库系统中的 B 树索引、Hash 索引、Bloom 过滤器进行优化，之后小组成员对其进行了优化 [6] 并将其拓展到了多维索引 [7]。此外，该组在 2019 年 NeurIPS 会议上提出了学习式索引结构的评测基准 SOSD[8]，并研发了一个新的融入机器学习模型的数据处理系统 SageDB[9]。

本文将对该组在学习式索引结构上做的这一系列工作进行梳理并介绍其重点内容，以期一窥将机器学习融入数据库系统这一方向上的研究进展。文章在接下来的组织如下：本文在第 2 部分将介绍阅读后文需要哪些背景知识，第 3 部分将介绍四种学习式索引结构——学习式范围索引、点索引、存在性索引、多维索引，第 4 部分介绍 MIT 数据系统组提出的学习式索引结构的评测基准 SOSD，第 5 部分介绍该组研发的 SageDB 数据处理系统，最后在第 6 部分对全文进行总结。

[1] http://dsg.csail.mit.edu/

2 基础知识

阅读后文所需的基础知识主要包括数据库中常见的索引结构和机器学习的基础知识，后文对用到的基础知识会做简要的介绍，篇幅原因这里不再做详细的介绍，关于数据库索引结构的内容可以参考文献 [1, 3, 4, 10]，关于机器学习的内容可以参考文献 [11, 12]。

3 学习式索引结构

尽管索引结构在提升数据访问查询效率方面已经必不可少，但是传统数据库中的索引结构并没有充分利用数据的分布信息：**它们没有对底层数据的分布做特定的假设**。尽管对数据不做任何假设的索引结构会更加通用，但是对于有特定分布的数据，通用索引的查询性能可能就会比较低。如 [5] 中所举得例子，如果我们对一组连续的整数进行查询，如果它们顺序存储在磁盘上，使用 B 树查询需要 $O(log\ n)$ 的时间复杂度和 $O(n)$ 的空间复杂度，而使用键-地址映射函数只需要 $O(1)$ 的时间复杂度和 $O(1)$ 的空间复杂度，因为键本身就对应着偏移量。虽然这是一个相对极端的例子，但是对于很多具有特定分布的数据来说，知道数据的分布将有助于极大地优化索引的效率 [5]。

但是对于现实世界的数据，我们怎么才能在不用人工分析的情况下知道数据的分布呢？近年来迅速发展的机器学习技术为我们提供了一种解决方案。我们可以利用机器学习技术自动挖掘数据的模式来构建更加高效的索引结构，即我们所说的学习式索引结构（learned indexes）。

但是机器学习模型作为索引结构存在两个方面的问题：在语义方面，如何将数据库索引结构建模成机器学习问题，保证设计的模型提供和传统数据库组件相同的语义；在性能方面，一些性能强大的机器学习模型，如神经网络，往往是难以训练的，我们又如何将其高效地构建成索引结构。面对这两个问题，[5] 为我们做出了解答：在语义方面，索引结构本身其实就可以看作是学习式的模型，比如 B 树索引可以看作一个输入为关键字输出为有序数组中的位置（也可以是一个范围的开始位置）的模型，Hash 索引可以看作是一个输入为关键字输出为无序数组中位置的模型，而 Bloom 过滤器可以看作是一个二分类模型，用来判断一个关键字是否存在；在性能方面，虽然神经网络这样的机器学习模型需要花费很多资源来训练，但是硬件的发展使得这个问题能够有效缓解，如今的 CPU 都已经可以支持单指令流多数据流（Single Instruction Multiple Data，SIMD）技术，而我们也可以语料到不远的将来，许多笔记本和手机将配备上 GPU 甚至是 TPU，并且有研究估计 GPU 的性能到 2025 年还能够再提升 1000 倍 [13]，机器学习模型的训练效率问题将逐渐缓解，并且由于机器学习模型相较于传统的索引结构能够更加充分地利用这些新型的硬件资源，机器学习模型作为索引结构在未来反而具有更大的优势。

当然，虽然上文中已经论述了机器学习模型存在作为索引结构的可能，但是在一些细节上机器学习模型和传统的索引结构仍然存在很大的不同，比如 Bloom 过滤器不会出现假阴性（false negatives）的问题而机器学习模型往往不会有这种特点。因此，对于特定的任务，我们需要对机器学习式索引结构做特定的设计，在下面几个小节中，我们将分别对如何将机器学习模型用作范围索引、点索引、存在性索引和多维索引做更详细的介绍。

3.1 范围索引

3.1.1 机器学习模型作为范围索引

B 树（及其变种，如 B+ 树、B* 树）是数据库系统中最常用的支持范围查询的索引结构。我们可以将 B 树看作一个模型，给定键 K，它能够“预测”其在有序数组中的位置 pos[2]。为了提升时间和空间效率，我们一般不会为所有的键构建索引，而是每隔 n 个记录，比如一个页（page）中的第一个记录, 构建一个索引。这样，我们就能够把 B 树看作一个回归树（regression tree），它将一个输入的键映射到一个具有最小误差值（$min-error$）和最大误差值（$max-error$）的位置区间，在这里最小误差值是 0，最大误差值是页的大小（$pagesize$），如图1(a) 所示。如果需要查找的键存在的话，我们就能够保证可以在区间 ($pos-0$, $pos+pagesize$) 里找到它。当有新的数据加入的时候，B 树需要再次调整到平衡，这也很类似机器学习中的重训练（re-trained）过程。

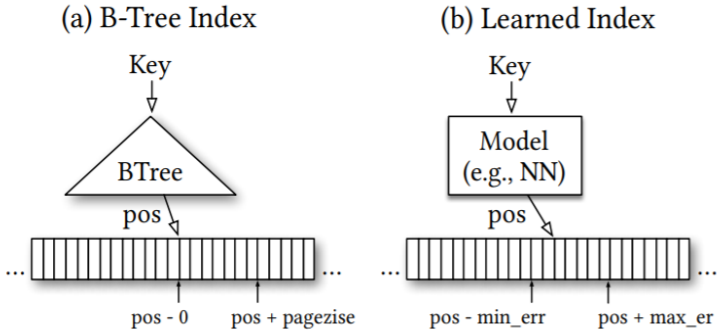

图 1: B-树可以看作是模型

这样看来，我们不一定非要用 B 树作为索引结构，因为别的模型如神经网络在训练后，我们也可以使得其误差在某个最小误差（min_err）和最大误差（max_err）的范围之中。更重要的是，我们甚至都不一定需要把误差固定在某一个范围内，因为我们假设数据是排序好的，因此只要我们能够提供一个大致的位置估计，在位置附近进行搜索将会很容易，因此我们完全可以将 B 树替换成其他的回归模型，如图1(b) 所示。同时，正如第 3 节一开始讲的例子中所展示的，将 B 树替换成别的机器学习模型，我们有机会降低搜索的时间复杂度和，同时我们还能够更好地利用新的硬件资源来加速。

[2]这里对模型做了比较强的假设，限定在内存中紧密排列的有序只读数组上。该组后面的工作对此限制条件进行了放宽。

3.1.2 RMI 模型

那么范围查询的索引结构应该替换成什么样的机器学习模型呢? 如 [5] 中所提到的，我们可以把数据的位置看作累积分布函数（Cumulative Distribution Function，CDF）。

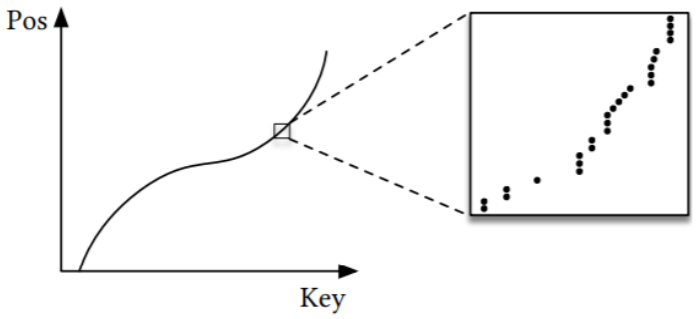

图 2: 索引可以看作是 CDF

如图 2 所示，我们可以建模数据的累积概率分布函数来预测其键的位置，抽象成的函数公式如式 (1) 所示：

$$p = F(Key) * N \tag{1}$$

其中 p 是估计的位置坐标，$F(Key)$ 是估计的累积分布函数，用来估计概率 $P(x \leq Key)$，N 是所有键的个数。

我们虽然可以用回归模型来拟合 CDF 函数，但是实验表明用简单的两层带 ReLU 激活函数的全连接网络来拟合后的效果并不是很好 [5]，主要原因之一是“最后一公里问题”——如图 2 所示，虽然 CDF 函数整体看上去很平滑，但是当我们放大某个点附近时，点的分布仍然很不规则。根据这个分析，我们可以想到，如果我们将数据进行拆分，每一小部分数据用对应的回归模型拟合，那么效果就会得到提升。

根据上述分析和思考，[5] 提出了递归模型索引（Recursive Model Index, RMI）。RMI 模型的示意图如图 3 所示:

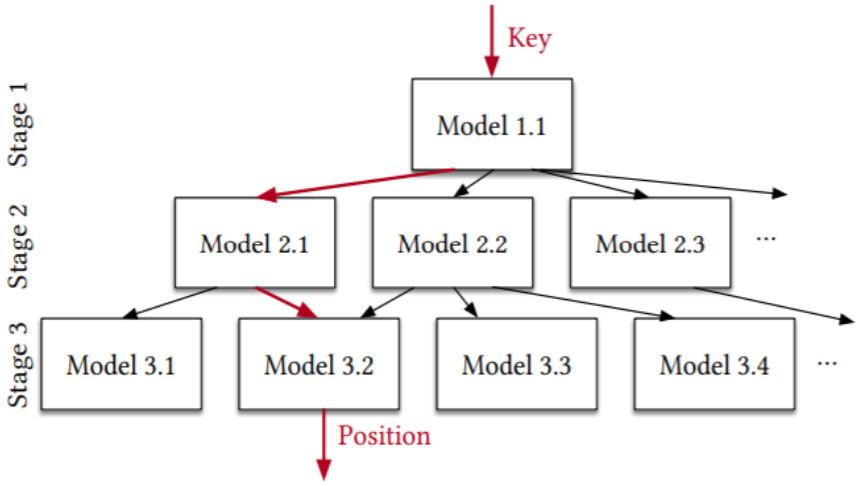

图 3: RMI 模型

- RMI 模型的预测：RMI 模型是一个层次化的树模型，每一层都有一个或者多个简单的回归模型，树的根节点和中间节点对应的模型起引导作用，用来选择下一层的模型，一直引导输入 K 到叶子结点，叶子结点根据其输入得到 K 对应的预测地址，然后在预测地址左右允许的误差范围内（误差范围在构建的时候通过计算整个数据集的最小和最大误差来得到）查找是否存在该键。RMI 的层数一般是 2-3 层，层数的递增表示拟合的数据范围的缩小，一直到叶子结点拟合最小范围的数据分布。这里的根节点往往采用简单神经网络，其他节点使用线性回归模型。如果数据分布过于复杂而无法拟合，叶子节点可以直接替换成传统的 B 树结构。

- RMI 模型的构建：RMI 模型构建需要先确定 RMI 的结构，比如有多少层，每一层有多少个模型，每个模型是什么模型（这里也可以用网格搜索或者其他模型结构搜索算法）。首先将所有数据分到根结点上，训练模型，然后根据模型的训练结果，将数据分到下层的不同结点上，然后下层结点用分配到自己结点的数据分别训练模型，然后继续向下迭代训练。这样就能够迭代训练整个模型。如果最后的训练误差超过提前设定的阈值，就将底层的模型改成 B 树。下面形式化地定义一下每层的损失函数：设我们的整个模型为 $f(x)$，其中 x 是键，$y \in [0, N)$ 是真实对应的位置，同时我们假设在 l 层有 M_l 个模型，l 层的第 k 个模型我们记为 $f_l^{(k)}$。我们在第 0 层训练一个模型 $f_0(x) \approx y$，其损失函数如式 (2)：

$$L_0 = \sum_{(x,y)} (f_0(x) - y)^2 \tag{2}$$

对于第 l 层，其损失定义为式 (3)：

$$L_l = \sum_{(x,y)} (f_l^{(\lfloor M_l f_{l-1}(x)/N \rfloor)}(x) - y) \tag{3}$$

其中我们迭代地计算 $f_{l-1}(x)$ 如式 (4)：

$$f_{l-1}(x) = f_{l-1}^{(\lfloor M_{l-1} f_{l-2}(x)/N \rfloor)}(x) \tag{4}$$

这样从上往下地依次迭代优化，我们就能够得到整个 RMI 模型。最后我们计算每个 K 对应的预测位置的最小最大误差，就能够得到索引在搜索的时候允许的误差范围。同时，这个误差范围也是评测 RMI 模型好坏的重要指标。

对于字符串的索引，RMI 模型还有其向量化变体：我们可以将字符串表示成一个向量，然后使得 RMI 模型的输入为一个向量，输出为字符串所在的位置。这是一个非常有趣的方向，这为将自然语言处理和计算机视觉等现在流行的深度学习技术融入数据库存储提供了新的探索方向，我们可以利用基于深度学习的表示来重新思考如何更加高效地存储字符串或者音频视频等数据，这也有助于推动多媒体信息检索技术的发展。

3.1.3 实验结果

Kraska 等人在三个数据集上对提出的索引结构与传统的 B 树索引进行了对比实验：一个网页日志数据集 Weblogs，一个地图数据集 Maps，和一个人造数据集 Lognormal。实验采用了两层的 RMI 模型，第一层是一个浅层神经网络，第二层是一些线性模型；实验采用了优化后的 B 树索引进行对比。对比实验的结果如图 4 所示，从图中可以看出，学习式的索引能够在减少两个数量级存储空间的情况下加速 1.5 到 3 倍，这充分体现了学习式索引的优势。作者做的其他几个对比实验也表明学习式索引能够在减少存储空间的同时提升检索效率 [5]。

		Map Data			Web Data			Log-Normal Data		
Type	**Config**	Size (MB)	Lookup (ns)	Model (ns)	Size (MB)	Lookup (ns)	Model (ns)	Size (MB)	Lookup (ns)	Model (ns)
Btree	page size: 32	52.45 (4.00x)	274 (0.97x)	198 (72.3%)	51.93 (4.00x)	276 (0.94x)	201 (72.7%)	49.83 (4.00x)	274 (0.96x)	198 (72.1%)
	page size: 64	26.23 (2.00x)	277 (0.96x)	172 (62.0%)	25.97 (2.00x)	274 (0.95x)	171 (62.4%)	24.92 (2.00x)	274 (0.96x)	169 (61.7%)
	page size: 128	13.11 (1.00x)	265 (1.00x)	134 (50.8%)	12.98 (1.00x)	260 (1.00x)	132 (50.8%)	12.46 (1.00x)	263 (1.00x)	131 (50.0%)
	page size: 256	6.56 (0.50x)	267 (0.99x)	114 (42.7%)	6.49 (0.50x)	266 (0.98x)	114 (42.9%)	6.23 (0.50x)	271 (0.97x)	117 (43.2%)
	page size: 512	3.28 (0.25x)	286 (0.93x)	101 (35.3%)	3.25 (0.25x)	291 (0.89x)	100 (34.3%)	3.11 (0.25x)	293 (0.90x)	101 (34.5%)
Learned Index	2nd stage models: 10k	0.15 (0.01x)	98 (2.70x)	31 (31.6%)	0.15 (0.01x)	222 (1.17x)	29 (13.1%)	0.15 (0.01x)	178 (1.47x)	26 (14.6%)
	2nd stage models: 50k	0.76 (0.06x)	85 (3.11x)	39 (45.9%)	0.76 (0.06x)	162 (1.60x)	36 (22.2%)	0.76 (0.06x)	162 (1.62x)	35 (21.6%)
	2nd stage models: 100k	1.53 (0.12x)	82 (3.21x)	41 (50.2%)	1.53 (0.12x)	144 (1.81x)	39 (26.9%)	1.53 (0.12x)	152 (1.73x)	36 (23.7%)
	2nd stage models: 200k	3.05 (0.23x)	86 (3.08x)	50 (58.1%)	3.05 (0.24x)	126 (2.07x)	41 (32.5%)	3.05 (0.24x)	146 (1.79x)	40 (27.6%)

图 4: 学习式索引 vs B 树索引

更多实验细节和实验结果请参考原论文 [5]。

3.1.4 RMI 模型改进

虽然 RMI 模型在实验结果上体现了一定的优势，但是其假设仍然太强，存在只能支持静态操作、数据划分不合理、仅考虑了均匀的查询负载等问题，现实中几乎无法使用，理论意义大于其实际意义。

针对索引只能支持静态操作（读操作）的问题，该组在 SIGMOD 2020 上发表了一篇新的论文 [6]。由于在原假设中，索引的底层数据是紧密的有序数组，这导致插入或者删除操作需要移动大量数据。并且，由于 RMI 结构是静态的树状层次结构，当叶子结点由于数据分布的改变需要重新训练时，其叶结点连接的所有上层结点都需要重新训练，这需要很高的训练成本，所以原模型只支持查找操作，并不支持插入、删除、更新等操作。针对此问题，论文 [6] 提出了新的设计思路，其提出模型的底层数据不再由有序数组紧密排列，各数据之间存在一定的空隙，使得插入操作无需移动过多的数据；其次，论文将 RMI 模型设计成动态的模型，当底层数据因为动态操作而过于紧密或者稀疏时，RMI 子模型能够自动进行拆分或者合并，子模型改变后，还会根据底层数据进行重新训练，从而提高查询的准确率。该设计将学习式的 B 树索引的实用化推进了一大步。

3.2 点索引

Hash 索引（如图 5(a) 所示）在支持数据库点查询的地位如同 B 树在支持数据库范围查询的地位。我们又如何将机器学习模型融入 Hash 索引呢？[5] 基于 RMI 模型和 CDF 提出了一种非常巧妙的模型设计。

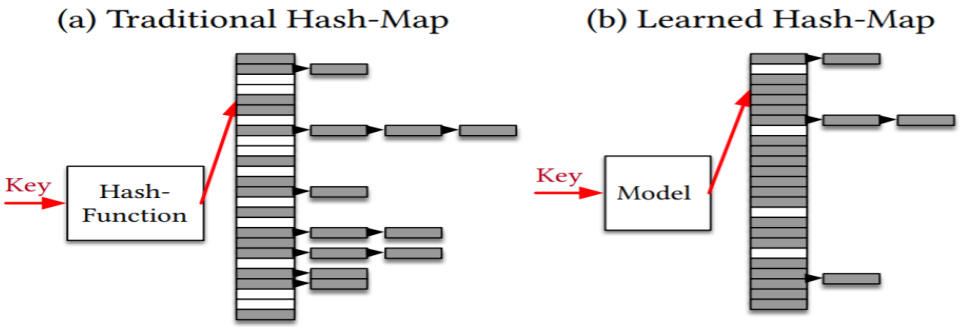

图 5: 传统 Hash 索引 vs 学习式 Hash 索引

类似于 RMI 模型中使用函数拟合 CDF 模型的方式，对于确定的 Hash 表大小 M，我们也可以使用类似 RMI 模型的方式拟合数据的 CDF，然后利用 M 对其进行缩放，

得到 Hash 表中的位置，如式 (5) 所示：

$$h(K) = F(K) * M \tag{5}$$

如果我们学习到的 CDF 比较好的话，经过 M 缩放后我们就能够将数据比较好地分散开。如果仍然存在冲突，那么便可以使用链表等方法来处理。整个学习式 Hash 索引结构如图 5(b) 所示。

[5] 的作者同样在前面所述的三个数据集上将学习式的 Hash 索引与传统的 Hash 索引做了对比，这里重点分析了其冲突量，实验结果如图 6 所示:

	% Conflicts Hash Map	% Conflicts Model	Reduction
Map Data	35.3%	07.9%	77.5%
Web Data	35.3%	24.7%	30.0%
Log Normal	35.4%	25.9%	26.7%

图 6: 传统 Hash 索引 vs 学习式 Hash 索引

实验结果表明学习式的 Hash 索引能够最多降低 77.5% 的冲突，体现了学习式 Hash 索引在避免冲突上的优势，但是由于缓存命中率等原因平均查找时间是传统 Hash 索引的 1.5 倍。

3.3 存在性索引

在数据库中高效地判断某个 key 是否存在是非常必要的，而 Bloom 过滤器就是这样一个高效的模型。Bloom 过滤器使用多个 Hash 函数，将一个键映射到一个给定位表 (bitmap) 中的多个位置，并将相应位置置为 1。对于一个新的键，如果经过多个 Hash 函数后，其对应的位表中的坐标值都是 1，那么我们判断 “这个键可能存在”；而如果有一个位置的值不是 1 而是 0，那么我们判断 “这个键一定不存在”。传统 Bloom 过滤器的示意图如图 7(a) 所示：
尽管有很多研究在尝试对 Bloom 过滤器进行优化 [14]，但是实际应用 Bloom 过滤器时由于要提升精度降低误判率仍然会消耗很大的内存空间来存储位表。

我们如何利用机器学习模型来对数据的模式进行利用,从而降低存储开销呢?Bloom 过滤器没有假负例的特点如何用机器学习模型实现呢？论文 [5] 中为我们提供了两种设计思路。

第一种方式是将机器学习模型建模成二分类模型，将键输入模型，我们能够预测其存在的概率，然后设定一个阈值来决定判断其是否存在。我们可以利用现有的数据当作正例样本，用机器学习的手段（如 GAN[15]）来生成负例样本，也就是实际不存在的样

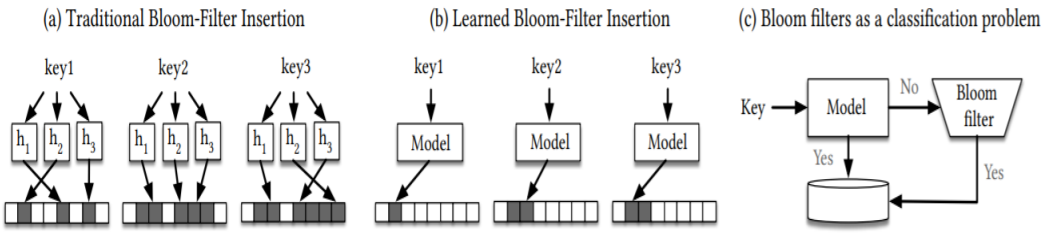

图 7: 传统 Bloom 过滤器 vs 学习式 Bloom 过滤器

本。我们利用正负样本来训练二分类模型无法保证假负例为 0，为了使得假负例彻底为 0，作者提出对于预测为 0 的样本，我们还需要将其输入传统的 Bloom 过滤器（backup filter），因此整个模型如图 7(c) 所示。此外，由于在实际模型中，如果假正率减小，假负率将会增大，假负率减小，假正率将会增大，为了使得模型在低假负率的同时把假正率限制在一个给定的范围内，我们还需要合理调节二分类模型的阈值。这样，由于模型参数所占空间相对较少，而备用 Bloom 过滤器由于输入量少了，其存储空间也能够显著减小，从而能够达到降低整个索引结构存储空间的效果。

另一种方式是构建新的 Hash 函数，如图 7(b) 所示，使得存在的键尽可能冲突，不存在的键也尽可能冲突，而存在的键和不存在的键尽可能没有冲突，这样就能够使得位图的存储大小减小。怎么构造这种函数呢，我们可以利用前一种方法中的二分类模型，由于模型输出可以是一个存在的概率 $p = f(key), p \in [0, 1]$，那么我们就能够利用这个概率，然后将其用位图的大小 m 缩放，即 $d = \lfloor f(x) * m \rfloor$，其中 d 是标记为 1 的位置坐标，这样当模型训练好之后，我们就能够使得位图的高位基本都表示存在的键，而底位基本都表示不存在的键，就能达到我们本段开始所希望的效果了。同样，为了使得假负率为 0，我们也可以加一个额外的 Bloom 过滤器，此时这个 Bloom 过滤器所占用的空间就比较小了。

论文中利用了谷歌的 ULR 数据来检测模型的效果，效果显示学习式的 Bloom 过滤器相较于传统的 Bloom 过滤器能够节约 30% 的存储空间。

3.4 多维索引

针对数据分析系统中常用的多维索引，SIGMOD 2020 上 Nathan 等人提出了一个学习式的多维度索引 Flood[7]，它能根据数据和负载自动优化索引结构和数据布局，从而提升查询的效率。

Flood 系统由两部分构成，一部分是离线进行的，用来得到最优的数据布局（layout）和索引，另一部分是一个在线的查询执行器，整个系统架构如图 8 所示。

在离线部分，如图 9 所示，Flood 系统将对每一个维度拟合一个 CDF 函数，然后

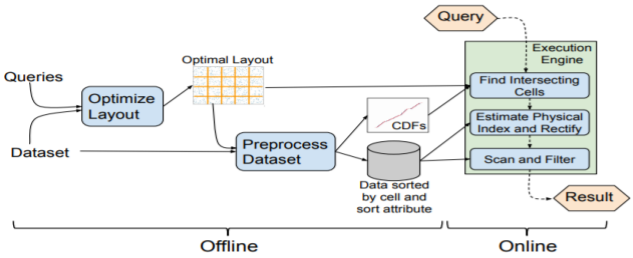

图 8: Flood 系统架构

通过将 CDF 划分，使得落在每个区间内有相同数量的数据。每个维度都进行划分之后，就会形成一个高维网格，我们就可以利用这个网格来做索引。

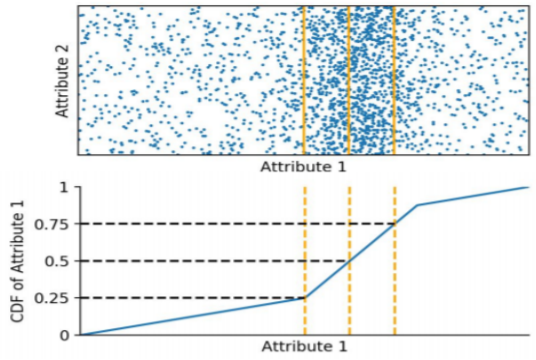

图 9: 利用 CDF 构建划分

但是只有网格的划分还不够，对于 d 维的数据，我们还需要选出一个维度用来排序，其他 $d-1$ 个维度来构成网格。如图 10 所示，对于一个 3 维空间的布局，为了加快查询效率，我们需要选择一个维度 z 来对每个 cell 内的数据进行排序，其他两个维度 x, y 用来做网格的划分。

这里选择合适的维度作为排序维度非常重要，[7] 提出了一种机器学习化的方法来优化布局（其实之前所述的基于 CDF 的维度划分，也需要用机器学习模型来自动确定划分的数目）。首先，Nathan 等人提出了 Flood 系统的查询执行流程，如图 11 所示，对于一个范围查询，我们可以将其看作超空间里的一个超立方体，我们首先选取出与这个超立方体有交集的 cell（图 11(1a)），然后对于每一个有交集的 cell，我们可以得到其物理地址的开始地址和结束地址（图 11(1b)），如果这个 cell 内的数据是排序好的，我们还能够对这个索引空间进一步减小（图 11(2)），之后我们会得到一个比较小的搜索空间，然后我们就能够对整个空间进行遍历和过滤了（图 11(3)）。

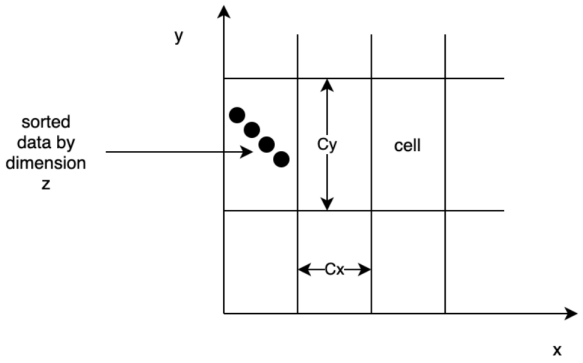

图 10: 三维数据布局

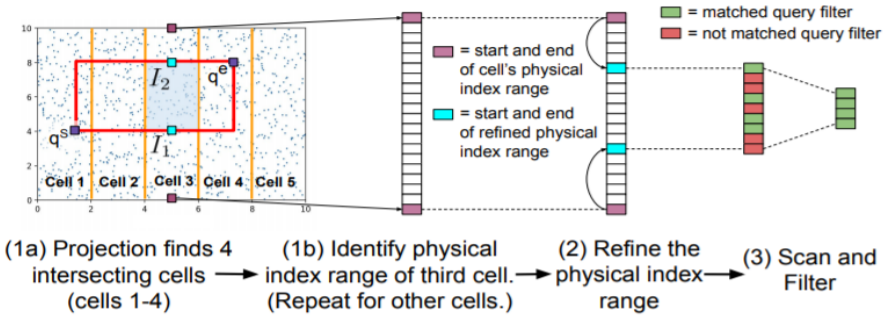

图 11: Flood 系统查询执行流程

有了查询的执行流程之后，我们如何得到最优的布局（用来排序的维度，和每个维度划分的数量）来使得查询延迟更小呢，Nathan 等人提出了一种代价模型（cost model），它定义了一个损失函数，如式 6 所示：

$$Time(D, q, L) = w_p N_c + w_r N_r + w_s N_s \tag{6}$$

其中 D 是给定数据集，q 是给定查询负载，L 是给定布局，w_p 是在一个 cell 上执行投影过程的平均时间，N_c 是落入查询超立方体中的 cell 的数目，w_r 是在一个 cell 上执行调整过程（refinement）的平均时间，w_s 是在一个 cell 上执行扫描操作的执行时间，N_s 是扫描点的数目。我们希望得到的布局 L 能够使得整个估计的查询执行时间最小，这便是我们的优化目标。

具体如何优化呢，首先我们先利用 RMI 模型得到每个维度的 CDF 估计，然后迭代地选取第 d 个维度作为排序维度，其余维度用来构成网格，然后对于每一个这样的

组合，我们以公式 6 作为目标函数，来优化每个维度划分的数目，其中的 w_p, w_r, w_c 我们利用一些数据的统计特征来建模，这样我们就能够使用梯度下降算法来选择出每个维度划分的数目，从而确定了整体的布局。整个训练的数据集和查询负载是需要提前手工设计或生成的，得到整个优化后的布局之后就布局就不再改变。

这样我们就得到了一个学习式的多维索引，实验表明学习式的索引结构能够达到三倍于传统聚簇索引的效果，相较于亚马逊的 Redshift's Z-encoding 算法，学习式的索引结构更能够达到 72 倍的效果。更多的实验细节和结果评测请参考原论文 [7]。

4 SOSD 基准评测

下面简单介绍一下 MIT 数据系统组提出的针对学习化索引结构的基准评测框架——SOSD：**S**earch **O**n **S**orted **D**ata Benchmark。该评测基准使用了多个真实的数据集和人工构造的数据集，并且实现了多个基准评测模型（baseline model），整个评测基准框架已经开源在 GitHub 上 [3]。

评测主要是针对基于内存的在有序数据上的查询索引，评测基准提供的数据集的 CDF 图像如图 12 所示。

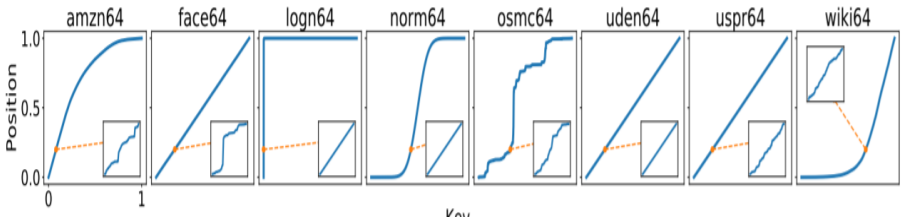

图 12: 数据集的 CDF

主要评测的技术类别如图 13 所示。

category	technique	layout
on-the-fly	BinarySearch (BS)	array
	InterpolationSearch (IS)	array
	TIP	array
aux. index	RadixBinarySearch (RBS)	array
approx. CDF	RadixSpline (RS)	array
	RMI	array
index	ART	custom
	B-tree	custom
	FAST	custom

图 13: 评测的技术类别

评测的实验结果如图 14 所示，其中绿色表示比较好的性能，黄色表示一般的性能，红色表示比较差的性能。从图中可以看出，学习式的索引结构确实存在着一定优势。更多细节请参考原论文 [8] 以及开源代码。

[3] https://github.com/learnedsystems/sosd

	ART	B-tree	BS	FAST	IS	RBS	RMI	RS	TIP
amzn32	n/a	529	773	**244**	4604	325	264	275	731
face32	**187**	524	771	229	1285	312	274	386	964
logn32	n/a	522	765	294	n/a	471	**97.0**	105	744
norm32	191	522	771	229	10257	355	**71.7**	**70.9**	884
uden32	102	521	771	228	**39.8**	333	54.2	64.2	176
uspr32	n/a	524	771	230	469	301	**153**	200	400
size overhead	47%	16%	0%	123%	0%	< 1%	3%	< 1%	0%
amzn64	n/a	601	804	n/a	4736	387	266	**288**	759
face64	391	592	784	n/a	1893	**337**	**334**	461	1232
logn64	309	597	784	n/a	n/a	753	179	**120**	454
norm64	266	592	785	n/a	10510	405	**71.5**	**70.5**	862
osmc64	n/a	599	785	n/a	95076	492	**402**	437	7186
uden64	112	592	784	n/a	**43.4**	344	54.3	53.9	193
uspr64	287	591	785	n/a	449	313	**169**	214	428
wiki64	n/a	608	802	n/a	7846	364	**222**	**218**	1019
size overhead	25%	16%	0%	n/a	0%	< 1%	3%	< 1%	0%

图 14: 查询延迟对比

5 SageDB 数据处理系统

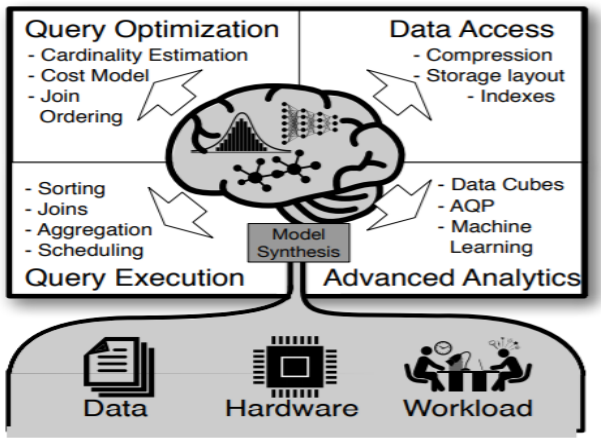

图 15: SageDB 示意图

最后我们简单介绍一下该组开发的一个新的学习式的数据处理系统 SageDB。SageDB 系统是一个基于机器学习的新型的数据库系统，它主要面向 OLAP 操作，对 OLTP 操作也能够做一定程度的改善。SageDB 利用机器学习方法根据数据的特征自动对整个系统进行优化，其在论文 [9] 中表示，“基于机器学习的组件可以完全取代数据库系统的核心组件，例如索引结构、排序算法，甚至是查询执行器”。整个 SageDB 系统的示意

图如图 15 所示。

SageDB 对传统数据库中多个核心组件进行了改进。在存储方面，其主要利用前文所述的 RMI 模型对多种常用的索引结构进行改进；在查询执行方面，论文提出了一种新的学习式的排序算法，核心思想是利用 CDF 函数将数据进行粗略地排序，然后再进行精调，整个学习式的排序算法如图 16 所示；学习的模型也可用于改善连接操作，例如，考虑一个具有两个存储连接列和一个模型列的合并连接，我们可以使用模型跳过不会连接的数据；作者还尝试了工作负载感知调度程序，使用图神经网络实现了基于强化学习的调度系统；在查询优化方面，作者最初的实验从传统的成本模型开始，并通过学习进行改进，结果表明模型质量可以得到改善，但要获得大幅收益，则需要对基数估算进行重大改进。论文中还提出了一些可以进行机器学习化的其他方向，如近似查询处理、预测建模和工作负载等。这些优化都取得了不错的效果。其他内容请参考论文 [9] 以及网页 [16]。

Algorithm 1 Learned sorting algorithm

Input a - the array to be sorted
Input F - the CDF model for the distribution of a
Input m - the over-allocation parameter
Output o - the sorted version of array a

```
1:  procedure LEARNED-SORT(a, F, m)
2:      o ← [∞] * (a.length * m)
3:      s ← {}
4:      // STEP 1: Approximate ordering
5:      for i in a do
6:          pos ← F(i) * a.length * m
7:          if o[pos] = ∞ then
8:              o[pos] ← i
9:          else
10:             s ← s ∪ {i}
11:     // STEP 2: Touch-up
12:     INSERTION-SORT(o)
13:     QUICKSORT(s)
14:     // STEP 3: Merging
15:     return MERGE-AND-REMOVE-EMPTY(o, s)
```

图 16: 学习式的排序算法

6 总结

本文主要调研了机器学习模型在数据库索引结构中的应用，以期一窥当前流行的机器学习技术和数据库系统方向的结合方法。

通过调研机器学习式的索引结构模型、评测指标和新的数据处理系统，我们可以看到，机器学习技术确实为数据库方面的研究打开了一扇新的大门，机器学习技术自动挖掘数据模式的能力有望能够全面提高传统数据库系统的性能。当然，从上文的论述中也可以看出，目前这仍然是一个不太成熟的方向，一些模型的假设过于严格，实验数据集的设置也不是那么合理，在一些数据集上学习式的索引结构也不一定比传统的索引结构性能更好 [17]，一些并发控制和分布式一致性等操作仍然没有解决（当然有其他论文提出了一些解决方案）。但是正如我们所看到的，MIT 数据系统组的这一系列工作从一定程度上证明了机器学习化数据库系统的可行性，同时给予了我们很多新的探索方向。同时，随着新硬件的发展，基于机器学习模型的设计会拥有更大的优势，这也使得数据库研究人员难以小看这一探索方向。

本文只是作为这一方向的入门级调研，更多研究还是需要深入论文和代码。一篇还不错的比较新的学习式数据库系统的综述文章是 [18]，尽管有些细节并不是非常严谨。

致谢

感谢杜云滔同学的本门课程课程报告的 $Latex$ 设计，本文的设计基本仿照其格式。本文在调研的过程中参考了简书、知乎等媒体上的文章，非常感谢其作者们，篇幅原因不再一一致谢。

参考文献

[1] 杨晨孟小峰, 马超红. 机器学习化数据库系统研究综述. 计算机研究与发展, 56(9):1803, 2019.

[2] Wei Wang, Meihui Zhang, Gang Chen, H. V. Jagadish, Beng Chin Ooi, and Kian-Lee Tan. Database meets deep learning. *ACM SIGMOD Record*, 45(2):17–22, Sep 2016.

[3] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation*. 2000.

[4] A. Silberschatz. Database system concepts 6th edition. 2010.

[5] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD ' 18, page 489–504, New York, NY, USA, 2018. Association for Computing Machinery.

[6] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, and et al. Alex: An updatable adaptive learned index. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, May 2020.

[7] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. Learning multi-dimensional indexes. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, May 2020.

[8] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. Sosd: A benchmark for learned indexes, 2019.

[9] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. Sagedb: A learned database system. 2019.

[10] Ming Gao. Notes on algorithms of data science and engineering. `http://dase.ecnu.edu.cn/mgao/teaching/DataSci_2020_Spring/reference/BOOK_20190217.pdf`, 2019.

[11] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 1 edition, 2007.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[13] Moore law is dead but gpu will get 1000x faster by 2025. https://www.nextbigfuture.com/2017/06/moore-law-is-dead-but-gpu-will-get-1000x-faster-by-2025.html.

[14] Michael Mitzenmacher. Compressed bloom filters. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, PODC ' 01, page 144–150, New York, NY, USA, 2001. Association for Computing Machinery.

[15] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014.

[16] Machine learning for systems. http://dsg.csail.mit.edu/mlforsystems/.

[17] Don't throw out your algorithms book just yet: Classical data structures that can outperform learned indexes. https://dawn.cs.stanford.edu//2018/01/11/index-baselines/.

[18] Du XY Chai MK, Fan J. Learnable database systems: Challenges and opportunities(in chinese). *Journal of Software*, pages 806–830, Mar 2020.