THE CASE FOR LEARNED INDEX STRUCTURE

SUMMARY

作者：李靖东

了解确切的数据分布可以高度优化数据库系统使用的几乎所有索引，而机器学习为学习反映数据模式和相关性的模型提供了机会，通过机器学习能够以较低的工程成本自动生成特制的索引结构，在文中这些通过学习生成的索引被称为 learned indexes。

机器学习模型的问题：
1. 不能提供传统索引所具有的语义保证
2. 最强大的机器学习模型（神经网络）通常被认为生成的代价很昂贵。

作者的结论：
1. B-tree 索引可以被看作是一个模型，将关键字作为输入，并预测其所在的位置，Bloom-filter 可以被看作是一个二分类器，基于一个关键字，预测关键字是否存在于某个集合中。尽管会存在一些细微而重要的差异，但是作者认为可以通过新的学习技术和简单的辅助结构来解决这些差异
2. 作者推测笔记本电脑和手机将很快会拥有 GPU 或 TPU，而 CPU-SIMD/GPU/TPUs 的功能将会越来越强大，因为相比于普通的指令集，神经网络所使用的受限的（并行）数学运算符更容易扩展，因此，执行神经网络的高成本在未来可能实际上可以忽略不计

这篇文章的重点在于 read-only 负载分析

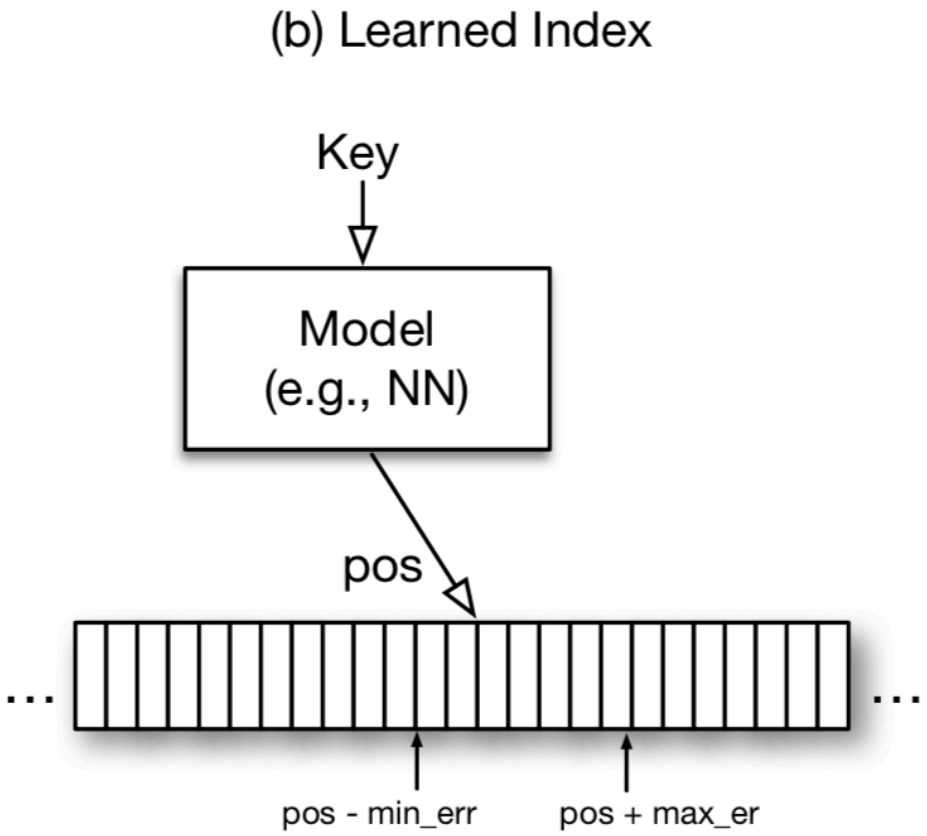
● Range Index

数据库系统通常使用 B-Tree 或 B+-Tree 来实现 range index
本例考虑基于内存数据库已排序主键的 B-tree 索引
假设:对已排序的数组进行逻辑分页，而不是位于不同存储区域中的物理页面

标准数据库的模式：查询时给定一个 key（或一些定义 range 的 keys），B-Tree 会索引到包含该 key 的对应范围的叶子节点，在叶子节点内对 key 进行搜索。如果该 key 在索引中存在，就会得到其对应的 position。这里 position 代表指向逻辑页的一个 offset 或 pointer 。一般在一个逻辑页内的 records 会用一个 key 来 index。

学习模型：B 树在机器学习术语中是一个回归树：它将一个关键字映射到区间 pospred - minerr, pospred + maxerr（minerr 期望为 0，maxerr 期望为页的大小）

并保证在该地区域可以找到关键字。因此，可以用其他类型的机器学习模型（包括深度学习模型）代替索引，只要它们也能够提供类似 min-err 和 max-err 的有力保证。



数据库模型的优势和机器学习模型的不足：

1. B-tree 插入和查找的代价有限，且对缓存的利用很好
2. B-tree 能够将关键字映射到不连续的内存和磁盘页
3. 如果查找的关键字不在集合中，特定的模型（不是单调递增的）可能会返回不在区间范围内的位置

学习模型的优势：

1. 神经网络具有强大的泛化能力，如果数据具有特殊性，那么用模型代替 B-Tree 就能减少查询复杂度
2. 神经网络的优点是能够学习各种各样的数据分布，混合和数据的模式及特性

方案：min-err 和 max-err 的计算由现有的数据（现有存储的（key，pos））计算出每组数的 pospred 和 pos 的正负差，然后取最大的正差为 max-err 最小的负差为 min-err，如果关键字不在索引中则不考虑。

其实 Range index 实际上是描述一个关键字到一个有序数组的映射关系，而这样一个映射关系有效近似于数据的累积分布函数（CDF），那么可以这么定义这个函数：$p = CDF(key) * N$，p 是 postion，CDF(x)是Prob(X ≤ key),N 是所有关

键字的总数，那么一个区间 X 的概率和为：$\mathrm{CDF}(key2) - CDF(key1) = Prob(key1 \leq X \leq key2)$。

第一次模型尝试：

模型：两层全连接神经网络（每层 32 个 neurons + ReLU acitivation）
训练集：200M 的 web-server log records
x 是 timstamp，label 是对应 B-Tree 索引到的 position

结果：
latency 大约是 80000 ns > 不用索引 > B 树索引 300ns
throughput 是每秒 1250 次预测

结果不好的原因：

1. 这里用了 Tensorflow 做训练和预测，Tensorflow 是为较大规模的神经网络做的优化，并不适用于这里的小网络，并且 python 因为用 swig 做胶水层会造成一定的 latency 开销。
2. 一般而言，B 树或者决策树在用很少的操作拟合数据方面表现的非常好，因为它们使用简单的 if 语句递归地分割空间。而用神经网络模型能够很快地学习出一个做地还不错的模型，但是很难做到局部数据上的准确（和 overfitting 有关）。为了保证精度，我们要训练更多次迭代，并且使用更深的网络来训练。
3. 典型的机器学习优化目标是减小平均错误率，然而对于索引来说，我们不只是需要预测关键字最可能在的位置，还需要确切的找到它。
4. B-Tree 是缓存高效的：能把上层节点放入 cache，并且在需要的时候访问其它页。而对于标准的神经网络模型，需要把模型都加载到内存中

方案：

1. The Learning Index Framework (LIF)

可以看作是一个索引优化系统，给定一套索引数据，LIF 自动生成多组超参数，并且自动地优化并测试。LIF 可以即时的学习简单的模型（线性回归模型），对于复杂的模型（神经网络）任然依赖于 TensorFlow。然而，相比于使用一个训练好的 Tensorflow 模型，LIF 自动的抽取模型中所有的权重，并基于模型的参数产生有效的用 C++编写的索引结构。

尽管使用 XLA 的 TensorFlow 支持代码编译，但它的重点主要放在大规模计算上，其中模型的执行时间是微秒或毫秒级。相比之下，LIF 的代码生成专注于小型模型，因此必须消除 Tensorflow 中不必要的开销和工具，这里利用[21]中的想法。因此，能够执行 30 纳秒级的简单模型。

2. The Recursive Model Index

全连接神经网络模型虽然可以从整体上较好地拟合出整体数据分布，但由于每个数据个体本身的随机性，却很难同时在局部也准确拟合出个体数据，文中把这个问题称作 the last-mile accuracy 问题。对于一个 100M 条记录的数据集，用单个简单的模型把误差控制在 10K 并不困难，然后对于局部数据，再用另外的模型进行预测把误差从 10K 降低到 100。

因此这里的思路就是用混合专家网络（Mixtures of Experts）[51]来解决 **the last-mile accuracy** 问题， MoE 的想法是能否把数据 partition，然后对于不同的数据，用不同的模型（称 Expert）来进行拟合。MoE 有个很好的性质：虽然在训练的时候由于模型很多，参数也很多。但是在预测时候模型的参数并不随着模型复杂度的增加而增加，因此计算开销也不回增加，因为总是能 Gating Network 找出几个相应的 Experts 来做预测。

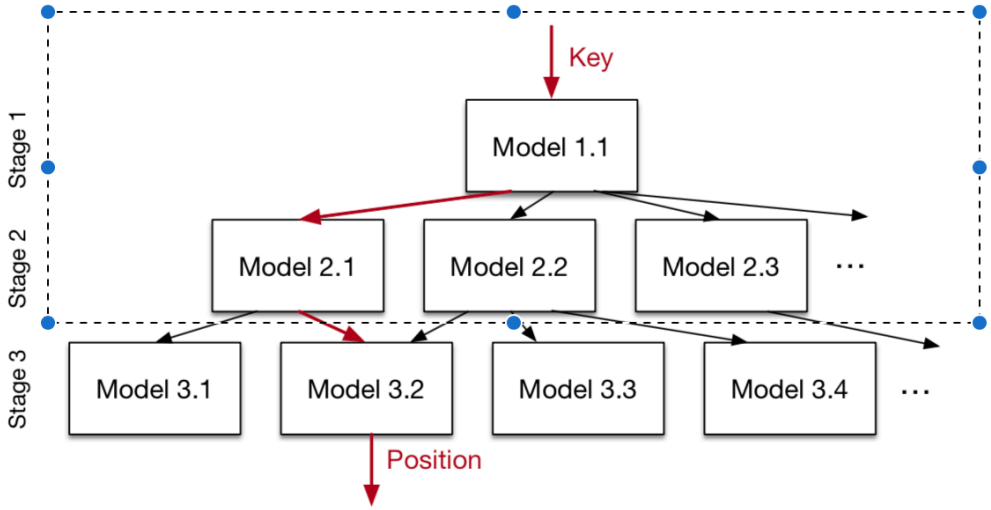


Figure 3: Staged models

3. Standard error-based search strategies

因为关键字的准确位置在预测位置的最小误差和最大误差范围内，依靠这个优势能够比传统的二分查找更快找到关键字。

模型二分查找：默认的查询策略，将传统二分查找的中间点设置为模型的预测值。

偏查找：对模型二分查找进行了修改，在每次迭代中不平均分割从中间到左右位置的范围，新的中间值取决于由最后阶段模型的标准偏差σ，如果关键字大于中间，则新的中间位置为$\min(\text{middle}+\sigma, \frac{\text{middle}+\text{right}}{2})$

偏四元查找：在每次迭代中不会只选择一个新的中间值来进行二进制搜索，而是三个新的数据点，如果 CPU 能够从主存储器中并行获取多个数据地址，那么这种策略可能表现会更好，但据称实际上这种策略与二进制搜索相当[8]。定义最初的三个四元搜索中点为 $pos-\sigma, pos, pos+\sigma$，如果位置的预测更为准确的话，这样做的结果会更好。

新的结果：

B-Trees ： 实现类似于 stx :: btree，但有进一步的 cache-line 优化，在一个小的 benckmark 上和 FAST[36]相比很有竞争力

Learning index：2 阶段 RMI 模型　第一阶段没有隐藏层 第二阶段 线性模型

Learning index complex：2 阶段 RMI 模型　第一阶段 2 个隐藏层，层宽 8 或者 16
第二阶段 线性模型

学习的所有索引模型都使用 LIF 编译

使用 Intel-E5 CPU，32GB RAM，查找超过 30M 次，重复 4 次

加载时间没有考虑（详细 参考 3.6）

数据集：

1. Weblogs 数据集包含 200 多万个日志条目，是多年来对某个主流大学网站的请求记录，并对所有唯一的时间戳进行了索引。包含了由课程安排，周末，假期，部门活动等引起

的非常复杂的时间模式，这些都是难以学习的。

2. Maps[46] 数据集是全世界 200M 由人维护的场所（博物馆，咖啡店等），用经度建索引，位置的经度相对线性，相比 Weblogs 数据集更加规则。
3. Web-document 数据集是由大型互联网公司的真实产品组成的大型网络索引的 10M 非连续文档 ID 组成。
4. Lognormal 为了测试索引如何处理重尾分布，我们生成了一个从对数正态分布采样得到的 190M 不同值的人造数据集，其中 μ = 0 和 σ = 2。这些值被放大到整数。这些数据是高度非线性的，这使 CDF 更难以使用神经网络来学习。

Type	Config	Search	Total (ns)	Model (ns)	Search (ns)	Speedup	Size (MB)	Size Savings	Model Err ± Err Var.
Btree	page size: 16	Binary	280	229	51	6%	104.91	700%	4 ± 0
	page size: 32	Binary	274	198	76	4%	52.45	300%	16 ± 0
	page size: 64	Binary	277	172	105	5%	26.23	100%	32 ± 0
	page size: 128	Binary	265	134	130	0%	13.11	0%	64 ± 0
	page size: 256	Binary	267	114	153	1%	6.56	-50%	128 ± 0
Learned Index	2nd stage size: 10,000	Binary	98	31	67	-63%	0.15	-99%	8 ± 45
		Quaternary	101	31	70	-62%	0.15	-99%	8 ± 45
	2nd stage size: 50,000	Binary	85	39	46	-68%	0.76	-94%	3 ± 36
		Quaternary	93	38	55	-65%	0.76	-94%	3 ± 36
	2nd stage size: 100,000	Binary	82	41	41	-69%	1.53	-88%	2 ± 36
		Quaternary	91	41	50	-66%	1.53	-88%	2 ± 36
	2nd stage size: 200,000	Binary	86	50	36	-68%	3.05	-77%	2 ± 36
		Quaternary	95	49	46	-64%	3.05	-77%	2 ± 36
Learned Index Complex	2nd stage size: 100,000	Binary	157	116	41	-41%	1.53	-88%	2 ± 30
		Quaternary	161	111	50	-39%	1.53	-88%	2 ± 30

Figure 4: Map data: Learned Index vs B-Tree

Type	Config	Search	Total (ns)	Model (ns)	Search (ns)	Speedup	Size (MB)	Size Savings	Model Err ± Err Var.
Btree	page size: 16	Binary	285	234	51	9%	103.86	700%	4 ± 0
	page size: 32	Binary	276	201	75	6%	51.93	300%	16 ± 0
	page size: 64	Binary	274	171	103	5%	25.97	100%	32 ± 0
	page size: 128	Binary	260	132	128	0%	12.98	0%	64 ± 0
	page size: 256	Binary	266	114	152	2%	6.49	-50%	128 ± 0
Learned Index	2nd stage size: 10,000	Binary	222	29	193	-15%	0.15	-99%	242 ± 150
		Quaternary	224	29	195	-14%	0.15	-99%	242 ± 150
	2nd stage size: 50,000	Binary	162	36	126	-38%	0.76	-94%	40 ± 27
		Quaternary	157	36	121	-40%	0.76	-94%	40 ± 27
	2nd stage size: 100,000	Binary	144	39	105	-45%	1.53	-88%	21 ± 14
		Quaternary	138	38	100	-47%	1.53	-88%	21 ± 14
	2nd stage size: 200,000	Binary	126	41	85	-52%	3.05	-76%	12 ± 7
		Quaternary	122	39	83	-53%	3.05	-76%	12 ± 7
Learned Index Complex	2nd stage size: 100,000	Binary	218	89	129	-16%	1.53	-88%	4218 ± 15917
		Quaternary	213	91	122	-18%	1.53	-88%	4218 ± 15917

Figure 5: Web Log Data: Learned Index vs B-Tree

Type	Config	Search	Total (ns)	Model (ns)	Search (ns)	Speedup	Size (MB)	Size Savings	Model Err ± Err Var.
Btree	page size: 16	Binary	285	233	52	9%	99.66	700%	4 ± 0
	page size: 32	Binary	274	198	77	4%	49.83	300%	16 ± 0
	page size: 64	Binary	274	169	105	4%	24.92	100%	32 ± 0
	page size: 128	Binary	263	131	131	0%	12.46	0%	64 ± 0
	page size: 256	Binary	271	117	154	3%	6.23	-50%	128 ± 0
Learned Index	2nd stage size: 10,000	Binary	178	26	152	-32%	0.15	-99%	17060 ± 61072
		Quaternary	166	25	141	-37%	0.15	-99%	17060 ± 61072
	2nd stage size: 50,000	Binary	162	35	127	-38%	0.76	-94%	17013 ± 60972
		Quaternary	152	35	117	-42%	0.76	-94%	17013 ± 60972
	2nd stage size: 100,000	Binary	152	36	116	-42%	1.53	-88%	17005 ± 60959
		Quaternary	146	36	110	-45%	1.53	-88%	17005 ± 60959
	2nd stage size: 200,000	Binary	146	40	106	-44%	3.05	-76%	17001 ± 60954
		Quaternary	148	45	103	-44%	3.05	-76%	17001 ± 60954
Learned Index Complex	2nd stage size: 100,000	Binary	178	110	67	-32%	1.53	-88%	8 ± 33
		Quaternary	181	111	70	-31%	1.53	-88%	8 ± 33

Figure 6: Synthetic Log-Normal: Learned Index vs B-Tree

	Config	Total (ns)	Model (ns)	Search (ns)	Speedup	Size (MB)	Size Savings	Std. Err ± Err Var.
Btree	page size: 32	1247	643	604	-3%	13.11	300%	8 ± 0
	page size: 64	1280	500	780	-1%	6.56	100%	16 ± 0
	page size: 128	1288	377	912	0	3.28	0	32 ± 0
	page size: 256	1398	330	1068	9%	1.64	-50%	64 ± 0
Learned Index	1 hidden layer	1605	503	1102	25%	1.22	-63%	104 ± 209
	2 hidden layers	1660	598	1062	29%	2.26	-31%	42 ± 75
Hybrid Index t=128	1 hidden layer	1397	472	925	8%	1.67	-49%	46 ± 29
	2 hidden layers	1620	591	1030	26%	2.33	-29%	38 ± 28
Hybrid Index t=64	1 hidden layer	1220	440	780	-5%	2.50	-24%	41 ± 20
	2 hidden layers	1447	556	891	12%	2.79	-15%	34 ± 20
Learned QS	1 hidden layer	1155	496	658	-10%	1.22	-63%	104 ± 209

Figure 7: String data: Learned Index vs B-Tree

	Binary Search		Biased Search		Quaternary Search	
	Total	Search	Total	Search	Total	Search
NN 1 hidden layer	1605	1102	1301	801	1155	658
NN 2 hidden layer	1660	1062	1338	596	1216	618

Figure 8: Learned String Index with different Search Strategies

实验过程：

其中 Weblogs、Maps 和 Lognormal 数据使用整数来当索引的，而 Web-documents 是用字符串来当索引的（因为不连续）。下面分别列出了四组数据的实验效果。在每组数据集的实验中，论文以 pagesize=128 作为 baseline，分别给出了 Learned Index 与 B-Tree Index 索引时间（Model 列表示搜索 position 的时间，Search 列表示搜索 record 的时间）的对比、索引内存占用的对比以及索引误差。在最后一组 Web-document 数据的实验里，文中对比了不同的搜索策略，因为对于 string 的 binary search 会更耗时，所以上文提出的搜索策略能够更明显地提高效率。论文对第二层里不同的 experts 数目做了不同的实验。总的来说，在索引的内存占用上大大有了显著减少。在索引时间上，在特定的配置下也能有一些提升。

未来的挑战和方向：

目前只是着眼于只读内存数据库系统的索引结构，作者认为即使没有任何重大的修改，当前的设计已经可以替代数据仓库中使用的索引结构，因为这些索引结构可能每天只更新一次，或者应用于 BigTable [18] 在 SStable 合并过程中，B-Tree 被批量的创造。

1. 中间数据的插入：B-tree 可以保证$O(\log n)$查找和插入成本，如果新插入的数据与学到的 CDF 有近似的模式，那么数据索引的建立和插入都是O(1)，如果插入数据的分布发生变化，不在本文的考虑范围，作者提到了 delta-index[49]，和一些优化方案[33]
2. 在本节中，作者假定数据存储在一个连续的块中。但是，特别是对于存储在磁盘上的数据的索引，将数据分割成存储在磁盘上不同区域中的较大页面，是相当常见的。为此，CDF 模型不再成立，作者提出利用 RMI 结构，通过对学习过程的小修改，简单的将偏移量存储到模型中，另一种方式是在<first_key，disk-position>的形式下添加附加的翻译表，索引结构的其余部分保持不变。但是，这个想法只有在磁盘页面非常大的情况下才会起作用

显然，需要更多的调查来更好地理解基于磁盘的 learning-index 的影响。与此同时，节省空间和速度的好处使其成为未来工作的一个非常有趣的路径。

● Point Index

Point Index是为了搜索出key对应的position

标准数据库模型：Hashmap。
学习模型：Hashmap 天然就是一个模型，输入是 key 输出是 position。

数据库模型的不足：对于 Hashmap 来讲，为了减少冲突，往往需要使用比 key 本身数目大很多的 slots 数目（论文中提到在 Google 的 dense-hashmap 有 78%额外的 slots）。对于冲突，虽然有 linked-list 或者 secondary probing 的方式来处理，但是它们的开销都不算小。即使用 sparse-hashmap 的方式来减小空间开销，搜索速度也会相比下降 3-7 倍。

学习模型的优势：学习模型能通过数据分布达到比传统方法更高的空间利用率，此外还可以将模型放到GPU/TPU上，可以减轻模型在计算hash函数上花费的代价。

方案：

1. 拟合累计分布函数 CDF 对学习 Hashmap 也起作用：h(K) = F(K) * M，M 是 hash-map 的目标大小，其中想训练出 F 去拟合数据的 CDF 函数。由于 point index 不需要保证 records 连续（不需要考虑 maxerr 和 minerr），所以训练出的模型 F 拟合 CDF 更容易得到满足。如果这时仍有少量的冲突，可以用 linked-list 的方式来处理。
2. Hash-map 模型的插入操作与正常 hash 的插入操作相同：将键与 h（k）进行 hash，并将项插入到返回的位置。如果位置已经被占用，则由 hash-map 的实现决定如何处理冲突。这意味着，只要插入数据与现有数据类似，hash 函数就可以保持其效率。但是，如果分布发生变化，那么模型可能需要按照前面所述进行再培训。

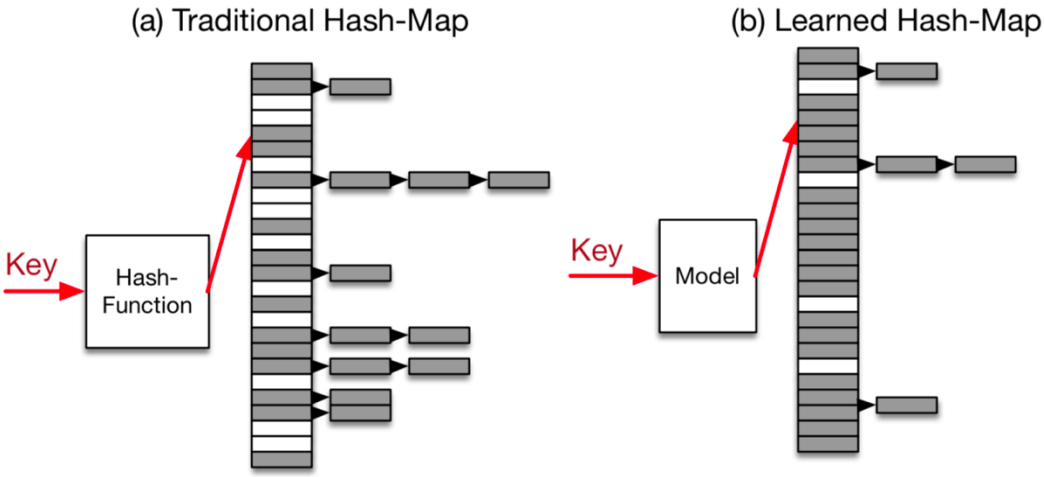


Figure 9: Traditional Hash-map vs Learned Hash-map

实验结果：

数据集：Map，Weblog，LogNormal

learning index：两层线性 experts 的 RMI 模型（最底层 100K 个 experts）

传统 hash-map：随机 hash 函数，该函数只使用两个乘法，3 个移位和 3 个异或

Dataset	Slots	Hash Type	Search Time (ns)	Empty Slots	Space Improvement
Map	75%	Model Hash	67	0.63GB (05%)	-20%
		Random Hash	52	0.80GB (25%)	
	100%	Model Hash	53	1.10GB (08%)	-27%
		Random Hash	48	1.50GB (35%)	
	125%	Model Hash	64	2.16GB (26%)	-6%
		Random Hash	49	2.31GB (43%)	
Web Log	75%	Model Hash	78	0.18GB (19%)	-78%
		Random Hash	53	0.84GB (25%)	
	100%	Model Hash	63	0.35GB (25%)	-78%
		Random Hash	50	1.58GB (35%)	
	125%	Model Hash	77	1.47GB (40%)	-39%
		Random Hash	50	2.43GB (43%)	
Log Normal	75%	Model Hash	79	0.63GB (20%)	-22%
		Random Hash	52	0.80GB (25%)	
	100%	Model Hash	66	1.10GB (26%)	-30%
		Random Hash	46	1.50GB (35%)	
	125%	Model Hash	77	2.16GB (41%)	-9%
		Random Hash	46	2.31GB (44%)	

Figure 10: Model vs Random Hash-map

结论：learned index 能达到更少的（~70%）empty slots 而搜索时间和原本 Hashmap 相近，这大大地提高了内存使用率。若 learned index 不能很好地学出 CDF（比如冲突很多），系统可以回滚到传统 Hashmap。

● Existence Index

用于测试一个元素是否是一个集合的成员

标准数据库模型：Bloom-Filter
学习模型：Binary Classification （略微修改，保证 FNR=0）

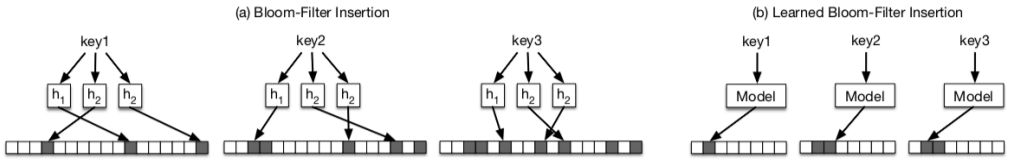


Figure 12: Bloom-filters with learned hash-functions

数据库模型的不足：Bloom-Filter 能天然杜绝 false negatives 的存在，但有 false positive 的存在。为了降低 false positive，比如对于一个 100M 记录的数据，Bloom-Filter 通常需要比 records 本身大 14 倍的 bits 来达到 0.1%的 false positive 准确度。虽然存在一些提升 Bloom-Filter 效率的尝试[43]，但是效果有待观察。

方案：learned index 目标和之前讨论的 B-Tree 及 Hashmap 都不一样。之前想学习出现 keys 的分布规律，而这里对于存在的 keys 我们却不需要区别对待。相反，想通过模型学出存在的 keys 和不存在的 keys 间的差异。换句话讲，期望存在的 keys 对应索引的 positions 尽量在一起，不存在的 keys 对应索引的 positions 尽量在一起，而存在 key 和不存在 key 索引出来的 position 尽量分开。同时，learned index 应该满足尽量低的 FPR 和内存占用且保证 FNR 为 0。

1. 可以把 existence index 看成是一个 binary classification 问题。考虑到建模字符串，可以使用 RNN 或 CNN 模型来做分类器。在预测的时候，模型会得到输入 key 是存在的可能性，需要给定一个 threshold（τ）来做出实际判断。然而，这样训练出来的不同模型有其对应的 FNR 和 FPR。我们可以通过 threshold 来降低 FPR，但 FPR 降低 FNR 会增加, 因为对于一个训练好的模型, 调整 τ 整个 False Rate 是不变的(FR = FPR+FNR), 这破坏了系统的 Semantic Guarantees。论文解决的思路非常简单：对于模型要给出 negative 预测的 keys，借助（rollback）传统的 bloom filter 来保证这个特性。由于这时的 bloom filter 只针对< threshold 的数据，因此总体来说 learned index 可以很好地优化内存。即使 FNR = 50%，理论上我们可以把内存优化到一半。这样一来， threshold 的选取就只需要考虑 FPR 了。
2. 还是把问题看成一个 binary classification 问题。训练得到了模型 f 以后，因为它满足不了 FNR=0，我们可以使用 f 作为 hash 函数，因为它将查询映射到 $[0,1]$范围，定义函数$d(p) = \lfloor mp \rfloor$，其中 m 是 bit array 的大小，这样$d(f(x))$就能作为 bloom-filter 中的 hash 函数，f 的优势是能够将查询的关键字和非关键字尽量分开。可以看到，两种方式都把上一段提到的目标当成一个 binary classification 问题，然后为了 Semantic Guarantees，都是结合 Bloom-filter 本身来实现的，只是在不同 level 的混合。

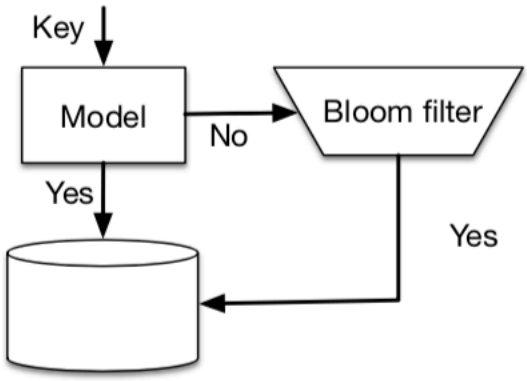


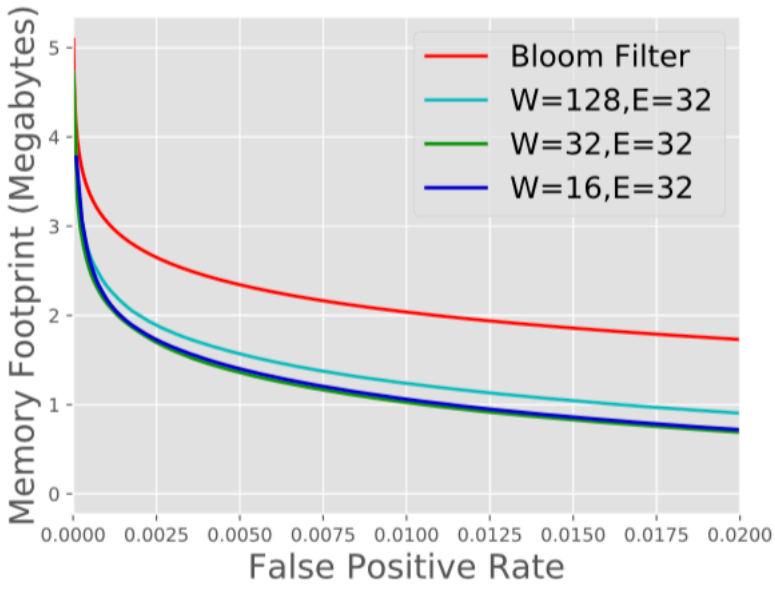
Figure 11: Bloom filters as a classification problem

实验结果：

数据集：google 报告的 1.7M 个钓鱼网站 URL 当成是 key set，一些随机的有效 URL 和会被误认为是钓鱼网站的白名单 URL 组成了 negative set。

Learning index：a character-level RNN (GRU [19], in particular)

Bloom filter：普通的 Bloom filter



模型越精确，Bloom-filter 大小就能减少，学习到的模型没必要和 Bloom-filter 功能相同，比如 WHOIS 数据和 IP 信息加入模型，能提高准确性，随之就可以减少 Bloom-filter 的大小，并保持 FNR=0

相关工作

B-Trees 和变种:现有的索引优化都没有从数据分布中学习来实现更紧凑的索引结构或性能收益。与此同时，与混合索引一样，将现有的基于硬件的索引优化策略与学习模型更紧密地结合起来，可以进一步提升性能。一些现有的压缩技术与 learning index 是相辅相成的，

可以帮助进一步提高效率。例如，字典压缩可以看作是一种 embedding（即将一个字符串 2 表示为一个唯一的整数），并且已经在我们的 web-doc 实验中使用。与本文最相关的是 A-Trees [24]，BF-Trees [12]和 B-Tree 插值搜索[27]。 BF-Tree 使用 B +树来存储关于数据集一个区域的信息，而不是索引个别键。但是，BF-Tree 中的叶节点是 bloom-filter，并且不近似于 CDF。相反，A-Trees 使用分段线性函数来减少 B 树中叶节点的数量，[27]建议在 B 树页内使用插值搜索。然而，学习索引更进一步，并建议使用学习模型替换整个索引结构。最后，像 Hippo [60]，Block Range Indexes [53]和 Small Materialized Aggregates（SMAs）[44]这样的稀疏索引都存储了关于值的范围信息，但是又不利用数据分布的基本属性。

更好的 Hash 函数：在 hash-maps 和 hash 函数[40,57,56,48]的研究上已经有很多工作,值得注意的是有人是用神经网络作为 hash 函数[55,57,32].然而这项工作和学习模型不同的是为了建立更有效的 hash-maps，它侧重于将高纬空间映射到一个更小的空间，为了相似搜索[57],或者是为机器学习创造更好的特征，又称 feature hashing[58]。可能最接近作者的思想：构建一个更好 hash 函数，的是[55]，它尝试使用神经网络来学习更强大的加密 hash 函数。然而，它的目标仍然不同于我们将关键字明确地通过 hash-map 映射到有限的 slots 集中。

Bloom-Filters：existence index 是建立在现有的 Bloom-Filter[22,11]之上的，然后作者提出一个增强分类的 Bloom-filter，和具有特殊 hash 函数的 Bloom-filter。

Succinct Data Structures：在 learning index 和 Succinct Data Structures 之间存在一个有趣的联系，尤其是 rank-select dictionaries，像 wavelet trees[31,30]。Learning index 可以通过学习数据的分布来预测元素的位置，因此，在以较慢的操作为代价的同时可以获得较高的压缩率，在理论上是可以达到 H0 熵。因此，Succinct Data Structures 可能可以给未来的 learned index 提供框架

CDF 建模：文中提到的 range index 和 point index 都和 CDF 函数密切相关，已经在机器学习的社区[41]中被研究，并且有一些应用，例如排名[34]，然而，大多数研究集中在对概率分布函数（PDF）进行建模，留下许多关于如何有效建模 CDF 的问题。

Mixture of Experts：作者的 RMI 架构 follow 了一系列为数据子集建立专家模式的工作，随着神经网络的发展，这也变得越来越普遍，并表现出更多的用途[51]。正如在文章中所展示的那样，这种架构能将模型大小和模型计算分隔开，从而实现执行代价不那么昂贵的复杂模型。

结论和未来工作

learning index 利用被索引的数据分布能获得显著的好处，这为许多有趣的工作打开了大门 Multi-Dimensional Indexes：学习模型，尤其是神经网络最擅长抓住复杂高维关系。

Beyond Indexing：CDF 模型也能加快排序和连接速度，使用现有的 CDF 模型的 F 函数将记录粗略的按照顺序排列，然后通过类似插入排序的方式，修正到几乎完美的排序数据。

GPU/TPUs：GPU/TPUs 使得 learning index 的想法更加可行，但是它们也有相应的挑战，最重要的是高调用延迟，根据之前实验结果中展示的压缩率，可以合理的假设，所有的 learning index 都适合 GPU/TPUs，但是还是需要 2-3 毫秒去调用它们操作。与此同时机器学习加速器和 CPU 的集成越来越好[6,4]，使用批量请求技术能分摊调用的代价，所以调用延迟现在也不是一个真正的障碍。

附录：

4. Intel Xeon Phi. https://www.intel.com/content/www/us/en/products/ processors/xeon-phi/xeon-phi-processors.html.

6. NVIDIA NVLink High-Speed Interconnect. http://www.nvidia.com/object/nvlink. html.

10. S. Abu-Nimeh, D. Nappa, X. Wang, and S. Nair. A comparison of machine learning techniques for phishing detection. In *Proceedings of the anti-phishing working groups 2nd annual eCrime researchers summit*, pages 60–69. ACM, 2007.

11. K. Alexiou, D. Kossmann, and P.-A. Larson. Adaptive range filters for cold data: Avoiding trips to siberia. *Proc. VLDB Endow.*, 6(14):1714–1725, Sept. 2013.

12. M. Athanassoulis and A. Ailamaki. BF-tree: Approximate Tree Indexing. In *VLDB*, pages 1881–1892, 2014.

21. A.Crotty,A.Galakatos,K.Dursun,T.Kraska,C.Binnig,U.C¸etintemel,andS.Zdonik.Anarchitecture for compiling udf-centric workflows. *PVLDB*, 8(12):1466–1477, 2015.

22. B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 75–88, New York, NY, USA, 2014. ACM.

24. A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska. A-tree: A bounded approximate index structure. under submission, 2017.

27. G. Graefe. B-tree indexes, interpolation search, and skew. In *Proceedings of the 2Nd International Workshop on Data Management on New Hardware*, DaMoN '06, New York, NY, USA, 2006. ACM.

31. R. Grossi and G. Ottaviano. The wavelet trie: Maintaining an indexed sequence of strings in compressed space. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '12, pages 203–214, New York, NY, USA, 2012. ACM.

32. J. Guo and J. Li. CNN based hashing for image retrieval. *CoRR*, abs/1509.01354, 2015.

34. J. C. Huang and B. J. Frey. Cumulative distribution networks and the derivative-sum-product algorithm: Models and inference for cumulative distribution functions on graphs. *J. Mach. Learn. Res.*, 12:301–348, Feb. 2011.

40. W. Litwin. Readings in database systems. chapter Linear Hashing: A New Tool for File and Table Addressing., pages 570–581. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

41. M. Magdon-Ismail and A. F. Atiya. Neural networks for density estimation. In M. J. Kearns, S. A. Solla, and D. A. Cohn, editors, *Advances in Neural Information Processing Systems 11*, pages 522–528. MIT Press, 1999.

44. G. Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB*, pages 476–487, 1998.

48. S. Richter, V. Alvarez, and J. Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. *Proc. VLDB Endow.*, 9(3):96–107, Nov. 2015.
51. N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.
53. M. Stonebraker and L. A. Rowe. The Design of POSTGRES. In *SIGMOD*, pages 340–355, 1986.
55. M. Turcanik and M. Javurek. Hash function generation by neural network. In *2016 New Trends in Signal Processing (NTSP)*, pages 1–5, Oct 2016.
56. J.Wang,W.Liu,S.Kumar,andS.F.Chang.Learning to hash for indexing big data; a survey. *Proceedings of the IEEE*, 104(1):34–57, Jan 2016.
57. J. Wang, H. T. Shen, J. Song, and J. Ji. Hashing for similarity search: A survey. *CoRR*, abs/1408.2927, 2014.
60. J. Yu and M. Sarwat. Two Birds, One Stone: A Fast, Yet Lightweight, Indexing Scheme for Modern Database Systems. In *VLDB*, pages 385–396, 2016.