arXiv:1712.01208v3 [cs.DB] 30 Apr 2018

The Case for Learned Index Structures

Tim Kraska*
MIT
Cambridge, MA
kraska@mit.edu

Alex Beutel
Google, Inc.
Mountain View, CA
alexbeutel@google.com

Ed H. Chi
Google, Inc.
Mountain View, CA
edchi@google.com

Jeffrey Dean
Google, Inc.
Mountain View, CA
jeff@google.com

Neoklis Polyzotis
Google, Inc.
Mountain View, CA
npolyzotis@google.com

Abstract

Indexes are models: a B-Tree-Index can be seen as a model to map a key to the position of a record within a sorted array, a Hash-Index as a model to map a key to a position of a record within an unsorted array, and a BitMap-Index as a model to indicate if a data record exists or not. In this exploratory research paper, we start from this premise and posit that all existing index structures can be replaced with other types of models, including deep-learning models, which we term *learned indexes*. The key idea is that a model can learn the sort order or structure of lookup keys and use this signal to effectively predict the position or existence of records. We theoretically analyze under which conditions learned indexes outperform traditional index structures and describe the main challenges in designing learned index structures. Our initial results show, that by using neural nets we are able to outperform cache-optimized B-Trees by up to 70% in speed while saving an order-of-magnitude in memory over several real-world data sets. More importantly though, we believe that the idea of replacing core components of a data management system through learned models has far reaching implications for future systems designs and that this work just provides a glimpse of what might be possible.

1 Introduction

Whenever efficient data access is needed, index structures are the answer, and a wide variety of choices exist to address the different needs of various access patterns. For example, B-Trees are the best choice for range requests (e.g., retrieve all records in a certain time frame); Hash-maps are hard to beat in performance for single key look-ups; and Bloom filters are typically used to check for record existence. Because of their importance for database systems and many other applications, indexes have been extensively tuned over the past decades to be more memory, cache and/or CPU efficient [36, 59, 29, 11].

Yet, all of those indexes remain general purpose data structures; they assume nothing about the data distribution and do not take advantage of more common patterns prevalent in real world data. For example, if the goal is to build a highly-tuned system to store and query ranges of fixed-length records over a set of continuous integer keys (e.g., the keys 1 to 100M), one would not use a conventional B-Tree index over the keys since the key itself can be used as an offset, making it an $O(1)$ rather than $O(\log n)$ operation to look-up any key or the beginning of a range of keys. Similarly, the index memory size would be reduced

*Work done while author was affiliated with Google.

$O(n)$ to $O(1)$. Maybe surprisingly, similar optimizations are possible for other data patterns. In other words, knowing the exact data distribution enables highly optimizing almost any index structure.

Of course, in most real-world use cases the data do not perfectly follow a known pattern and the engineering effort to build specialized solutions for every use case is usually too high. However, we argue that machine learning (ML) opens up the opportunity to learn a model that reflects the patterns in the data and thus to enable the automatic synthesis of specialized index structures, termed **learned indexes**, with low engineering cost.

In this paper, we explore the extent to which learned models, including neural networks, can be used to enhance, or even replace, traditional index structures from B-Trees to Bloom filters. This may seem counterintuitive because ML cannot provide the semantic guarantees we traditionally associate with these indexes, and because the most powerful ML models, neural networks, are traditionally thought of as being very compute expensive. Yet, we argue that none of these apparent obstacles are as problematic as they might seem. Instead, our proposal to use learned models has the potential for significant benefits, especially on the next generation of hardware.

In terms of semantic guarantees, indexes are already to a large extent learned models making it surprisingly straightforward to replace them with other types of ML models. For example, a B-Tree can be considered as a model which takes a key as an input and predicts the position of a data record in a sorted set (the data has to be sorted to enable efficient range requests). A Bloom filter is a binary classifier, which based on a key predicts if a key exists in a set or not. Obviously, there exists subtle but important differences. For example, a Bloom filter can have false positives but not false negatives. However, as we will show in this paper, it is possible to address these differences through novel learning techniques and/or simple auxiliary data structures.

In terms of performance, we observe that every CPU already has powerful SIMD capabilities and we speculate that many laptops and mobile phones will soon have a Graphics Processing Unit (GPU) or Tensor Processing Unit (TPU). It is also reasonable to speculate that CPU-SIMD/GPU/TPUs will be increasingly powerful as it is much easier to scale the restricted set of (parallel) math operations used by neural nets than a general purpose instruction set. As a result the high cost to execute a neural net or other ML models might actually be negligible in the future. For instance, both Nvidia and Google's TPUs are already able to perform thousands if not tens of thousands of neural net operations in a single cycle [3]. Furthermore, it was stated that GPUs will improve $1000\times$ in performance by 2025, whereas Moore's law for CPUs is essentially dead [5]. By replacing branch-heavy index structures with neural networks, databases and other systems can benefit from these hardware trends. While we see the future of learned index structures on specialized hardware, like TPUs, this paper focuses entirely on CPUs and surprisingly shows that we can achieve significant advantages even in this case.

It is important to note that we do not argue to completely replace traditional index structures with learned indexes. Rather, **the main contribution of this paper is to outline and evaluate the potential of a novel approach to build indexes, which complements existing work and, arguably, opens up an entirely new research direction for a decades-old field.** This is based on the key observation that **many data structures can be decomposed into a learned model and an auxiliary structure** to provide the same semantic guarantees. The potential power of this approach comes from the fact that **continuous functions, describing the data distribution, can be used to build more efficient data structures or algorithms**. We empirically get very promising results when evaluating our approach on synthetic and real-world datasets for read-only analytical workloads. However, many open challenges still remain, such as how to handle write-heavy workloads, and we outline many possible directions for future work. Furthermore, we believe that we can use the same principle to replace other components and operations commonly used in (database) systems. If successful, the core idea of deeply embedding learned models into algorithms and data structures could lead to a radical departure from the way systems are currently developed.

The remainder of this paper is outlined as follows: In the next two sections we introduce the general

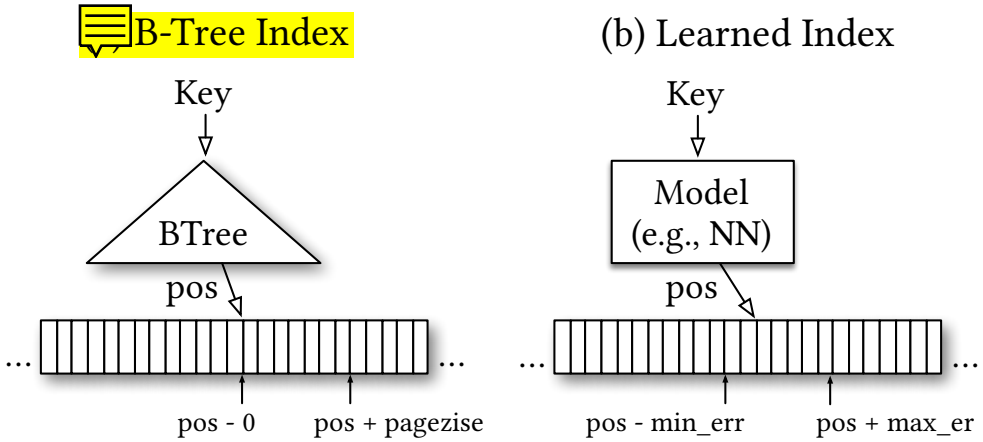

Figure 1: Why B-Trees are models

idea of learned indexes using B-Trees as an example. In Section 4 we extend this idea to Hash-maps and in Section 5 to Bloom filters. All sections contain a separate evaluation. Finally in Section 6 we discuss related work and conclude in Section 7.

2 Range Index

Range index structure, like B-Trees, are already models: given a key, they "predict" the location of a value within a key-sorted set. To see this, consider a B-Tree index in an analytics in-memory database (i.e., read-only) over the sorted primary key column as shown in Figure 1(a). In this case, the B-Tree provides a mapping from a look-up key to a position inside the sorted array of records with the guarantee that the key of the record at that position is the first key equal or higher than the look-up key. The data has to be sorted to allow for efficient range requests. This same general concept also applies to secondary indexes where the data would be the list of `<key, record_pointer>` pairs with the key being the indexed value and the pointer a reference to the record.[1]

For efficiency reasons it is common not to index every single key of the sorted records, rather only the key of every n-th record, i.e., the first key of a page. Here we only assume fixed-length records and logical paging over a continuous memory region, i.e., a single array, not physical pages which are located in different memory regions (physical pages and variable length records are discussed in Appendix D.2). Indexing only the first key of every page helps to significantly reduce the number of keys the index has to store without any significant performance penalty. Thus, the B-Tree is a model, or in ML terminology, a regression tree: it maps a key to a position with a min- and max-error (a min-error of 0 and a max-error of the page-size), with a guarantee that the key can be found in that region if it exists. Consequently, we can replace the index with other types of ML models, including neural nets, as long as they are also able to provide similar strong guarantees about the min- and max-error.

At first sight it may seem hard to provide the same guarantees with other types of ML models, but it is actually surprisingly simple. First, the B-Tree only provides the strong min- and max-error guarantee over the stored keys, not for all possible keys. For new data, B-Trees need to be re-balanced, or in machine learning terminology re-trained, to still be able to provide the same error guarantees. That is, for monotonic models the only thing we need to do is to execute the model for every key and remember the worst over- and

[1]Note, that against some definitions for secondary indexes we do not consider the `<key, record_pointer>` pairs as part of the index; rather for secondary index the data are the `<key, record_pointer>` pairs. This is similar to how indexes are implemented in key value stores [12, 21] or how B-Trees on modern hardware are designed [44].

under-prediction of a position to calculate the min- and max-error.[2] Second, and more importantly, the strong error bounds are not even needed. The data has to be sorted anyway to support range requests, so any error is easily corrected by a local search around the prediction (e.g., using exponential search) and thus, even allows for non-monotonic models. Consequently, we are able to replace B-Trees with any other type of regression model, including linear regression or neural nets (see Figure 1(b)).

Now, there are other technical challenges that we need to address before we can replace B-Trees with learned indexes. For instance, B-Trees have a bounded cost for inserts and look-ups and are particularly good at taking advantage of the cache. Also, B-Trees can map keys to pages which are not continuously mapped to memory or disk. All of these are interesting challenges/research questions and are explained in more detail, together with potential solutions, throughout this section and in the appendix.

At the same time, using other types of models as indexes can provide tremendous benefits. Most importantly, it has the potential to transform the $\log n$ cost of a B-Tree look-up into a constant operation. For example, assume a dataset with 1M unique keys with a value from 1M and 2M (so the value 1,000,009 is stored at position 10). In this case, a simple linear model, which consists of a single multiplication and addition, can perfectly predict the position of any key for a point look-up or range scan, whereas a B-Tree would require $\log n$ operations. The beauty of machine learning, especially neural nets, is that they are able to learn a wide variety of data distributions, mixtures and other data peculiarities and patterns. The challenge is to balance the complexity of the model with its accuracy.

For most of the discussion in this paper, we keep the simplified assumptions of this section: we only index an in-memory dense array that is sorted by key. This may seem restrictive, but many modern hardware optimized B-Trees, e.g., FAST [44], make exactly the same assumptions, and these indexes are quite common for in-memory database systems for their superior performance [44, 48] over scanning or binary search. However, while some of our techniques translate well to some scenarios (e.g., disk-resident data with very large blocks, for example, as used in Bigtable [23]), for other scenarios (fine grained paging, insert-heavy workloads, etc.) more research is needed. In Appendix D.2 we discuss some of those challenges and potential solutions in more detail.

2.1 What Model Complexity Can We Afford?

To better understand the model complexity, it is important to know how many operations can be performed in the same amount of time it takes to traverse a B-Tree, and what precision the model needs to achieve to be more efficient than a B-Tree.

Consider a B-Tree that indexes 100M records with a page-size of 100. We can think of every B-Tree node as a way to partition the space, decreasing the "error" and narrowing the region to find the data. We therefore say that the B-Tree with a page-size of 100 has a *precision gain* of $1/100$ per node and we need to traverse in total $log_{100}N$ nodes. So the first node partitions the space from $100M$ to $100M/100 = 1M$, the second from $1M$ to $1M/100 = 10k$ and so on, until we find the record. Now, traversing a single B-Tree page with binary search takes roughly 50 cycles and is notoriously hard to parallelize[3]. In contrast, a modern CPU can do 8-16 SIMD operations per cycle. Thus, a model will be faster as long as it has a better precision gain than $1/100$ per $50 * 8 = 400$ arithmetic operations. Note that this calculation still assumes that all B-Tree pages are in the cache. A single cache-miss costs 50-100 additional cycles and would thus allow for even more complex models.

Additionally, machine learning accelerators are entirely changing the game. They allow to run much more complex models in the same amount of time and offload computation from the CPU. For example,

[2]The model has to be monotonic to also guarantee the min- and max-error for look-up keys, which do not exist in the stored set.

[3]There exist SIMD optimized index structures such as FAST [44], but they can only transform control dependencies to memory dependencies. These are often significantly slower than multiplications with simple in-cache data dependencies and as our experiments show SIMD optimized index structures, like FAST, are not significantly faster.

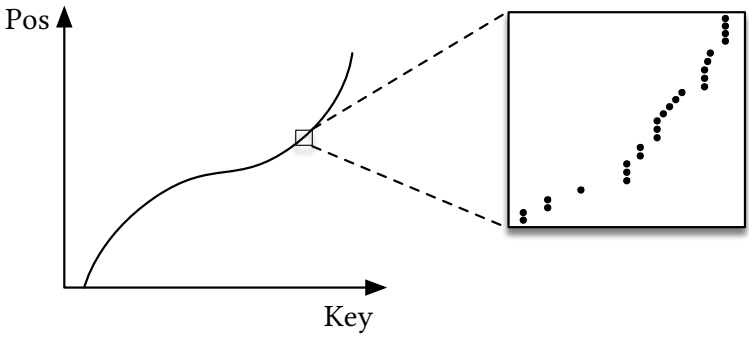

Figure 2: Indexes as CDFs

NVIDIA's latest Tesla V100 GPU is able to achieve 120 TeraFlops of low-precision deep learning arithmetic operations ($\approx 60,000$ operations per cycle). Assuming that the entire learned index fits into the GPU's memory (we show in Section 3.7 that this is a very reasonable assumption), in just 30 cycles we could execute 1 million neural net operations. Of course, the latency for transferring the input and retrieving the result from a GPU is still significantly higher, but this problem is not insuperable given batching and/or the recent trend to more closely integrate CPU/GPU/TPUs [4]. Finally, it can be expected that the capabilities and the number of floating/int operations per second of GPUs/TPUs will continue to increase, whereas the progress on increasing the performance of executing if-statements of CPUs essentially has stagnated [5]. Regardless of the fact that we consider GPUs/TPUs as one of the main reasons to adopt learned indexes in practice, in this paper we focus on the more limited CPUs to better study the implications of replacing and enhancing indexes through machine learning without the impact of hardware changes.

2.2 Range Index Models are CDF Models

As stated in the beginning of the section, an index is a model that takes a key as an input and predicts the position of the record. Whereas for point queries the order of the records does not matter, for range queries the data has to be sorted according to the look-up key so that all data items in a range (e.g., in a time frame) can be efficiently retrieved. This leads to an interesting observation: a model that predicts the position given a key inside a sorted array effectively approximates the cumulative distribution function (CDF). We can model the CDF of the data to predict the position as:

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

where p is the position estimate, $F(\text{Key})$ is the estimated cumulative distribution function for the data to estimate the likelihood to observe a key smaller or equal to the look-up key $P(X \leq \text{Key})$, and N is the total number of keys (see also Figure 2). This observation opens up a whole new set of interesting directions: First, it implies that indexing literally requires learning a data distribution. A B-Tree "learns" the data distribution by building a regression tree. A linear regression model would learn the data distribution by minimizing the (squared) error of a linear function. Second, estimating the distribution for a dataset is a well known problem and learned indexes can benefit from decades of research. Third, learning the CDF plays also a key role in optimizing other types of index structures and potential algorithms as we will outline later in this paper. Fourth, there is a long history of research on how closely theoretical CDFs approximate empirical CDFs that gives a foothold to theoretically understand the benefits of this approach [28]. We give a high-level theoretical analysis of how well our approach scales in Appendix A.

2.3 A First, Naïve Learned Index

To better understand the requirements to replace B-Trees through learned models, we used 200M web-server log records with the goal of building a secondary index over the timestamps using Tensorflow [9]. We trained a two-layer fully-connected neural network with 32 neurons per layer using ReLU activation functions; the timestamps are the input features and the positions in the sorted array are the labels. Afterwards we measured the look-up time for a randomly selected key (averaged over several runs disregarding the first numbers) with Tensorflow and Python as the front-end.

In this setting we achieved ≈ 1250 predictions per second, i.e., it takes $\approx 80,000$ nano-seconds (ns) to execute the model with Tensorflow, without the search time (the time to find the actual record from the predicted position). As a comparison point, a B-Tree traversal over the same data takes $\approx 300ns$ and binary search over the entire data roughly $\approx 900ns$. With a closer look, we find our naïve approach is limited in a few key ways:

1. Tensorflow was designed to efficiently run larger models, not small models, and thus, has a significant invocation overhead, especially with Python as the front-end.

2. B-Trees, or decision trees in general, are really good in overfitting the data with a few operations as they recursively divide the space using simple if-statements. In contrast, other models can be significantly more efficient to approximate the general shape of a CDF, but have problems being accurate at the individual data instance level. To see this, consider again Figure 2. The figure demonstrates, that from a top-level view, the CDF function appears very smooth and regular. However, if one zooms in to the individual records, more and more irregularities show; a well known statistical effect. Thus models like neural nets, polynomial regression, etc. might be more CPU and space efficient to narrow down the position for an item from the entire dataset to a region of thousands, but a single neural net usually requires significantly more space and CPU time for the "last mile" to reduce the error further down from thousands to hundreds.

3. B-Trees are extremely cache- and operation-efficient as they keep the top nodes always in cache and access other pages if needed. In contrast, standard neural nets require all weights to compute a prediction, which has a high cost in the number of multiplications.

3 The RM-Index

In order to overcome the challenges and explore the potential of models as index replacements or optimizations, we developed the learning index framework (LIF), recursive-model indexes (RMI), and standard-error-based search strategies. We primarily focus on simple, fully-connected neural nets because of their simplicity and flexibility, but we believe other types of models may provide additional benefits.

3.1 The Learning Index Framework (LIF)

The LIF can be regarded as an index synthesis system; given an index specification, LIF generates different index configurations, optimizes them, and tests them automatically. While LIF can learn simple models on-the-fly (e.g., linear regression models), it relies on Tensorflow for more complex models (e.g., NN). However, it never uses Tensorflow at inference. Rather, given a trained Tensorflow model, LIF automatically extracts all weights from the model and generates efficient index structures in C++ based on the model specification. Our code-generation is particularly designed for small models and removes all unnecessary overhead and instrumentation that Tensorflow has to manage the larger models. Here we leverage ideas from [25], which already showed how to avoid unnecessary overhead from the Spark-runtime. As a result,

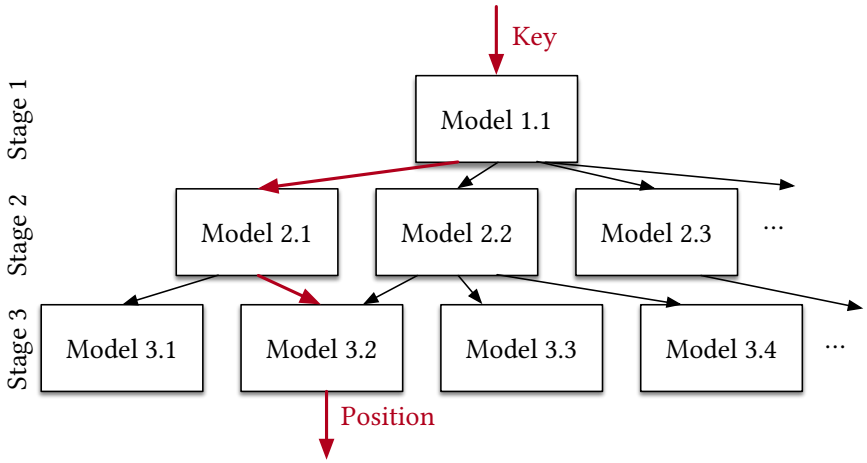

Figure 3: Staged models

we are able to execute simple models on the order of 30 nano-seconds. However, it should be pointed out that LIF is still an experimental framework and is instrumentalized to quickly evaluate different index configurations (e.g., ML models, page-sizes, search strategies, etc.), which introduces additional overhead in form of additional counters, virtual function calls, etc. Also besides the vectorization done by the compiler, we do not make use of special SIMD intrinisics. While these inefficiencies do not matter in our evaluation as we ensure a fair comparison by always using our framework, for a production setting or when comparing the reported performance numbers with other implementations, these inefficiencies should be taking into account/be avoided.

3.2 The Recursive Model Index

As outlined in Section 2.3 one of the key challenges of building alternative learned models to replace B-Trees is the accuracy for last-mile search. For example, reducing the prediction error to the order of hundreds from 100M records using a single model is often difficult. At the same time, reducing the error to 10k from 100M, e.g., a precision gain of $100 * 100 = 10000$ to replace the first 2 layers of a B-Tree through a model, is much easier to achieve even with simple models. Similarly, reducing the error from 10k to 100 is a simpler problem as the model can focus only on a subset of the data.

Based on that observation and inspired by the mixture of experts work [62], we propose the recursive regression model (see Figure 3). That is, we build a hierarchy of models, where at each stage the model takes the key as an input and based on it picks another model, until the final stage predicts the position. More formally, for our model $f(x)$ where x is the key and $y \in [0, N)$ the position, we assume at stage ℓ there are M_ℓ models. We train the model at stage 0, $f_0(x) \approx y$. As such, model k in stage ℓ, denoted by $f_\ell^{(k)}$, is trained with loss:

$$L_\ell = \sum_{(x,y)} (f_\ell^{(\lfloor M_\ell f_{\ell-1}(x)/N \rfloor)}(x) - y)^2 \qquad\qquad L_0 = \sum_{(x,y)} (f_0(x) - y)^2$$

Note, we use here the notation of $f_{\ell-1}(x)$ recursively executing $f_{\ell-1}(x) = f_{\ell-1}^{(\lfloor M_{\ell-1} f_{\ell-2}(x)/N \rfloor)}(x)$. In total, we iteratively train each stage with loss L_ℓ to build the complete model. One way to think about the different models is that each model makes a prediction with a certain error about the position for the *key* and that the prediction is used to select the next model, which is responsible for a certain area of the key-space to make a better prediction with a lower error. However, recursive model indexes do *not* have to be trees.

As shown in Figure 3 it is possible that different models of one stage pick the same models at the stage below. Furthermore, each model does not necessarily cover the same amount of records like B-Trees do (i.e., a B-Tree with a page-size of 100 covers 100 or less records).[4] Finally, depending on the used models the predictions between the different stages can not necessarily be interpreted as positions estimates, rather should be considered as picking an expert which has a better knowledge about certain keys (see also [62]).

This model architecture has several benefits: (1) It separates model size and complexity from execution cost. (2) It leverages the fact that it is easy to learn the overall shape of the data distribution. (3) It effectively divides the space into smaller sub-ranges, like a B-Tree, to make it easier to achieve the required "last mile" accuracy with fewer operations. (4) There is no search process required in-between the stages. For example, the output of *Model 1.1* is directly used to pick the model in the next stage. This not only reduces the number of instructions to manage the structure, but also allows representing the entire index as a sparse matrix-multiplication for a TPU/GPU.

3.3 Hybrid Indexes

Another advantage of the recursive model index is, that we are able to build mixtures of models. For example, whereas on the top-layer a small ReLU neural net might be the best choice as they are usually able to learn a wide-range of complex data distributions, the models at the bottom of the model hierarchy might be thousands of simple linear regression models as they are inexpensive in space and execution time. Furthermore, we can even use traditional B-Trees at the bottom stage if the data is particularly hard to learn.

For this paper, we focus on 2 types of models, simple neural nets with zero to two fully-connected hidden layers and ReLU activation functions and a layer width of up to 32 neurons and B-Trees (a.k.a. decision trees). Note, that a zero hidden-layer NN is equivalent to linear regression. Given an index configuration, which specifies the number of stages and the number of models per stage as an array of sizes, the end-to-end training for hybrid indexes is done as shown in Algorithm 1

Algorithm 1: Hybrid End-To-End Training

```
   Input: int threshold, int stages[], NN_complexity
   Data: record data[], Model index[][]
   Result: trained index
1  M = stages.size;
2  tmp_records[][];
3  tmp_records[1][1] = all_data;
4  for i ← 1 to M do
5  |   for j ← 1 to stages[i] do
6  |   |   index[i][j] = new NN trained on tmp_records[i][j];
7  |   |   if i < M then
8  |   |   |   for r ∈ tmp_records[i][j] do
9  |   |   |   |   p = index[i][j](r.key) / stages[i + 1];
10 |   |   |   |   tmp_records[i + 1][p].add(r);
11 for j ← 1 to index[M].size do
12 |   index[M][j].calc_err(tmp_records[M][j]);
13 |   if index[M][j].max_abs_err > threshold then
14 |   |   index[M][j] = new B-Tree trained on tmp_records[M][j];
15 return index;
```

Starting from the entire dataset (line 3), it trains first the top-node model. Based on the prediction of this top-node model, it then picks the model from the next stage (lines 9 and 10) and adds all keys which fall into

[4]Note, that we currently train stage-wise and not fully end-to-end. End-to-end training would be even better and remains future work.

that model (line 10). Finally, in the case of hybrid indexes, the index is optimized by replacing NN models with B-Trees if absolute min-/max-error is above a predefined threshold (lines 11-14).

Note, that we store the standard and min- and max-error for every model on the last stage. That has the advantage, that we can individually restrict the search space based on the used model for every key. Currently, we tune the various parameters of the model (i.e., number of stages, hidden layers per model, etc.) with a simple simple grid-search. However, many potential optimizations exists to speed up the training process from ML auto tuning to sampling.

Note, that hybrid indexes allow us to bound the worst case performance of learned indexes to the performance of B-Trees. That is, in the case of an extremely difficult to learn data distribution, all models would be automatically replaced by B-Trees, making it virtually an entire B-Tree.

3.4 Search Strategies and Monotonicity

Range indexes usually implement an $upper\_bound(key)$ [$lower\_bound(key)$] interface to find the position of the first key within the sorted array that is equal or higher [lower] than the look-up key to efficiently support range requests. For learned range indexes we therefore have to find the first key higher [lower] from the look-up key based on the prediction. Despite many efforts, it was repeatedly reported [8] that binary search or scanning for records with small payloads are usually the fastest strategies to find a key within a sorted array as the additional complexity of alternative techniques rarely pays off. However, learned indexes might have an advantage here: the models actually predict the position of the key, not just the region (i.e., page) of the key. Here we discuss two simple search strategies which take advantage of this information:

Model Biased Search: Our default search strategy, which only varies from traditional binary search in that the first *middle* point is set to the value predicted by the model.

Biased Quaternary Search: Quaternary search takes instead of one split point three points with the hope that the hardware pre-fetches all three data points at once to achieve better performance if the data is not in cache. In our implementation, we defined the initial three middle points of quaternary search as $pos - \sigma, pos, pos + \sigma$. That is we make a guess that most of our predictions are accurate and focus our attention first around the position estimate and then we continue with traditional quaternary search.

For all our experiments we used the min- and max-error as the search area for all techniques. That is, we executed the RMI model for every key and stored the worst over- and under-prediction per last-stage model. While this technique guarantees to find all existing keys, for non-existing keys it might return the wrong upper or lower bound if the RMI model is not monotonic. To overcome this problem, one option is to force our RMI model to be monotonic, as has been studied in machine learning [41, 71].

Alternatively, for non-monotonic models we can automatically adjust the search area. That is, if the found upper (lower) bound key is on the boundary of the search area defined by the min- and max-error, we incrementally adjust the search area. Yet, another possibility is, to use exponential search techniques. Assuming a normal distributed error, those techniques on average should work as good as alternative search strategies while not requiring to store any min- and max-errors.

3.5 Indexing Strings

We have primarily focused on indexing real valued keys, but many databases rely on indexing strings, and luckily, significant machine learning research has focused on modeling strings. As before, we need to design a model of strings that is efficient yet expressive. Doing this well for strings opens a number of unique challenges.

The first design consideration is how to turn strings into features for the model, typically called tokenization. For simplicity and efficiency, we consider an n-length string to be a feature vector $\mathbf{x} \in \mathbb{R}^n$ where $\mathbf{x}_i$ is the ASCII decimal value (or Unicode decimal value depending on the strings). Further, most ML models

		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%)

Figure 4: Learned Index vs B-Tree

operate more efficiently if all inputs are of equal size. As such, we will set a maximum input length N. Because the data is sorted lexicographically, we will truncate the keys to length N before tokenization. For strings with length $n < N$, we set $\mathbf{x}_i = 0$ for $i > n$.

For efficiency, we generally follow a similar modeling approach as we did for real valued inputs. We learn a hierarchy of relatively small feed-forward neural networks. The one difference is that the input is not a single real value x but a vector $\mathbf{x}$. Linear models $\mathbf{w} \cdot \mathbf{x} + \mathbf{b}$ scale the number of multiplications and additions linearly with the input length N. Feed-forward neural networks with even a single hidden layer of width h will scale $O(hN)$ multiplications and additions.

Ultimately, we believe there is significant future research that can optimize learned indexes for string keys. For example, we could easily imagine other tokenization algorithms. There is a large body of research in natural language processing on string tokenization to break strings into more useful segments for ML models, e.g., wordpieces in translation [70]. Further, it might be interesting to combine the idea of suffix-trees with learned indexes as well as explore more complex model architectures (e.g., recurrent and convolutional neural networks).

3.6 Training

While the training (i.e., loading) time is not the focus of this paper, it should be pointed out that all of our models, shallow NNs or even simple linear/multi-variate regression models, train relatively fast. Whereas simple NNs can be efficiently trained using stochastic gradient descent and can converge in less than one to a few passes over the randomized data, a closed form solution exists for linear multi-variate models (e.g., also 0-layer NN) and they can be trained in a single pass over the sorted data. Therefore, for 200M records training a simple RMI index does not take much longer than a few seconds, (of course, depending on how much auto-tuning is performed); neural nets can train on the order of minutes per model, depending on the complexity. Also note that training the top model over the entire data is usually not necessary as those models converge often even before a single scan over the entire randomized data. This is in part because we use simple models and do not care much about the last few digit points in precision, as it has little effect on indexing performance. Finally, research on improving learning time from the ML community [27, 72] applies in our context and we expect a lot of future research in this direction.

3.7 Results

We evaluated learned range indexes in regard to their space and speed on several real and synthetic data sets against other read-optimized index structures.

3.7.1 Integer Datasets

As a first experiment we compared learned indexes using a 2-stage RMI model and different second-stage sizes (10k, 50k, 100k, and 200k) with a read-optimized B-Tree with different page sizes on three different integer data sets. For the data we used 2 real-world datasets, (1) Weblogs and (2) Maps [56], and (3) a synthetic dataset, Lognormal. The Weblogs dataset contains 200M log entries for every request to a major university web-site over several years. We use the unique request timestamps as the index keys. This dataset is almost a worst-case scenario for the learned index as it contains very complex time patterns caused by class schedules, weekends, holidays, lunch-breaks, department events, semester breaks, etc., which are notoriously hard to learn. For the maps dataset we indexed the longitude of $\approx$ 200M user-maintained features (e.g., roads, museums, coffee shops) across the world. Unsurprisingly, the longitude of locations is relatively linear and has fewer irregularities than the Weblogs dataset. Finally, to test how the index works on heavy-tail distributions, we generated a synthetic dataset of 190M unique values sampled from a log-normal distribution with $\mu = 0$ and $\sigma = 2$. The values are scaled up to be integers up to 1B. This data is of course highly non-linear, making the CDF more difficult to learn using neural nets. For all B-Tree experiments we used 64-bit keys and 64-bit payload/value.

As our baseline, we used a production quality B-Tree implementation which is similar to the stx::btree but with further cache-line optimization, dense pages (i.e., fill factor of 100%), and very competitive performance. To tune the 2-stage learned indexes we used simple grid-search over neural nets with zero to two hidden layers and layer-width ranging from 4 to 32 nodes. In general we found that a simple (0 hidden layers) to semi-complex (2 hidden layers and 8- or 16-wide) models for the first stage work the best. For the second stage, simple, linear models, had the best performance. This is not surprising as for the last mile it is often not worthwhile to execute complex models, and linear models can be learned optimally.

Learned Index vs B-Tree performance: The main results are shown in Figure 4. Note, that the page size for B-Trees indicates the number of keys per page not the size in Bytes, which is actually larger. As the main metrics we show the size in MB, the total look-up time in nano-seconds, and the time to execution the model (either B-Tree traversal or ML model) also in nano-seconds and as a percentage compared to the total time in paranthesis. Furthermore, we show the speedup and space savings compared to a B-Tree with page size of 128 in parenthesis as part of the size and lookup column. We choose a page size of 128 as the fixed reference point as it provides the best lookup performance for B-Trees (note, that it is always easy to save space at the expense of lookup performance by simply having no index at all). The color-encoding in the speedup and size columns indicates how much faster or slower (larger or smaller) the index is against the reference point.

As can be seen, the learned index dominates the B-Tree index in almost all configurations by being up to $1.5 - 3\times$ faster while being up to two orders-of-magnitude smaller. Of course, B-Trees can be further compressed at the cost of CPU-time for decompressing. However, most of these optimizations are orthogonal and apply equally (if not more) to neural nets. For example, neural nets can be compressed by using 4- or 8-bit integers instead of 32- or 64-bit floating point values to represent the model parameters (a process referred to as quantization). This level of compression can unlock additional gains for learned indexes.

Unsurprisingly the second stage size has a significant impact on the index size and look-up performance. Using 10,000 or more models in the second stage is particularly impressive with respect to the analysis in §2.1, as it demonstrates that our first-stage model can make a much larger jump in precision than a single node in the B-Tree. Finally, we do not report on hybrid models or other search techniques than binary search for these datasets as they did not provide significant benefit.

Learned Index vs Alternative Baselines: In addition to the detailed evaluation of learned indexes against our read-optimized B-Trees, we also compared learned indexes against other alternative baselines, including third party implementations, under fair conditions. In the following, we discuss some alternative baselines and compare them against learned indexes if appropriate:

	Lookup Table w/ AVX search	FAST	Fixe-Size Btree w/ interpol. search	Multivariate Learned Index
Time	199 ns	189 ns	280 ns	105 ns
Size	16.3 MB	1024 MB	1.5 MB	1.5 MB

Figure 5: Alternative Baselines

Histogram: B-Trees approximate the CDF of the underlying data distribution. An obvious question is whether histograms can be used as a CDF model. In principle the answer is yes, but to enable fast data access, the histogram must be a low-error approximation of the CDF. Typically this requires a large number of buckets, which makes it expensive to search the histogram itself. This is especially true, if the buckets have varying bucket boundaries to efficiently handle data skew, so that only few buckets are empty or too full. The obvious solutions to this issues would yield a B-Tree, and histograms are therefore not further discussed.

Lookup-Table: A simple alternative to B-Trees are (hierarchical) lookup-tables. Often lookup-tables have a fixed size and structure (e.g., 64 slots for which each slot points to another 64 slots, etc.). The advantage of lookup-tables is that because of their fixed size they can be highly optimized using AVX instructions. We included a comparison against a 3-stage lookup table, which is constructed by taking every 64th key and putting it into an array including padding to make it a multiple of 64. Then we repeat that process one more time over the array without padding, creating two arrays in total. To lookup a key, we use binary search on the top table followed by an AVX optimized branch-free scan [14] for the second table and the data itself. This configuration leads to the fastest lookup times compared to alternatives (e.g., using scanning on the top layer, or binary search on the 2nd array or the data).

FAST: FAST [44] is a highly SIMD optimized data structure. We used the code from [47] for the comparison. However, it should be noted that FAST always requires to allocate memory in the power of 2 to use the branch free SIMD instructions, which can lead to significantly larger indexes.

Fixed-size B-Tree & interpolation search: Finally, as proposed in a recent blog post [1] we created a fixed-height B-Tree with interpolation search. The B-Tree height is set, so that the total size of the tree is 1.5MB, similar to our learned model.

Learned indexes without overhead: For our learned index we used a 2-staged RMI index with a multivariate linear regression model at the top and simple linear models at the bottom. We used simple automatic feature engineering for the top model by automatically creating and selecting features in the form of key, $log(key)$, key^2, etc. Multivariate linear regression is an interesting alternative to NN as it is particularly well suited to fit nonlinear patterns with only a few operations. Furthermore, we implemented the learned index outside of our benchmarking framework to ensure a fair comparison.

For the comparison we used the Lognormal data with a payload of an eight-byte pointer. The results can be seen in Figure 5. As can be seen for the dataset under fair conditions, learned indexes provide the best overall performance while saving significant amount of memory. It should be noted, that the FAST index is big because of the alignment requirement.

While the results are very promising, we by no means claim that learned indexes will always be the best choice in terms of size or speed. Rather, learned indexes provide a new way to think about indexing and much more research is needed to fully understand the implications.

3.7.2 String Datasets

We also created a secondary index over 10M non-continuous document-ids of a large web index used as part of a real product at Google to test how learned indexes perform on strings. The results for the string-based document-id dataset are shown in Figure 6, which also now includes hybrid models. In addition, we include our best model in the table, which is a non-hybrid RMI model index with quaternary search, named "Learned

	Config	Size(MB)	Lookup (ns)	Model (ns)
Btree	page size: 32	13.11 (4.00x)	1247 (1.03x)	643 (52%)
	page size: 64	6.56 (2.00x)	1280 (1.01x)	500 (39%)
	page size: 128	3.28 (1.00x)	1288 (1.00x)	377 (29%)
	page size: 256	1.64 (0.50x)	1398 (0.92x)	330 (24%)
Learned Index	1 hidden layer	1.22 (0.37x)	1605 (0.80x)	503 (31%)
	2 hidden layers	2.26 (0.69x)	1660 (0.78x)	598 (36%)
Hybrid Index	t=128, 1 hidden layer	1.67 (0.51x)	1397 (0.92x)	472 (34%)
	t=128, 2 hidden layers	2.33 (0.71x)	1620 (0.80x)	591 (36%)
	t= 64, 1 hidden layer	2.50 (0.76x)	1220 (1.06x)	440 (36%)
	t= 64, 2 hidden layers	2.79 (0.85x)	1447 (0.89x)	556 (38%)
Learned QS	1 hidden layer	1.22 (0.37x)	1155 (1.12x)	496 (43%)

Figure 6: String data: Learned Index vs B-Tree

QS" (bottom of the table). All RMI indexes used 10,000 models on the 2nd stage and for hybrid indexes we used two thresholds, 128 and 64, as the maximum tolerated absolute error for a model before it is replaced with a B-Tree.

As can be seen, the speedups for learned indexes over B-Trees for strings are not as prominent. Part of the reason is the comparably high cost of model execution, a problem that GPU/TPUs would remove. Furthermore, searching over strings is much more expensive thus higher precision often pays off; the reason why hybrid indexes, which replace bad performing models through B-Trees, help to improve performance.

Because of the cost of searching, the different search strategies make a bigger difference. For example, the search time for a NN with 1-hidden layer and biased binary search is $1102ns$ as shown in Figure 6. In contrast, our biased quaternary search with the same model only takes $658ns$, a significant improvement. The reason why biased search and quaternary search perform better is that they take the model error into account.

4 Point Index

Next to range indexes, Hash-maps for point look-ups play a similarly important role in DBMS. Conceptually Hash-maps use a hash-function to deterministically map keys to positions inside an array (see Figure 7(a)). The key challenge for any efficient Hash-map implementation is to prevent too many distinct keys from being mapped to the same position inside the Hash-map, henceforth referred to as a *conflict*. For example, let's assume 100M records and a Hash-map size of 100M. For a hash-function which uniformly randomizes the keys, the number of expected conflicts can be derived similarly to the birthday paradox and in expectation would be around 33% or 33M slots. For each of these conflicts, the Hash-map architecture needs to deal with this conflict. For example, separate chaining Hash-maps would create a linked-list to handle the conflict (see Figure 7(a)). However, many alternatives exist including secondary probing, using buckets with several slots, up to simultaneously using more than one hash function (e.g., as done by Cuckoo Hashing [57]).

However, regardless of the Hash-map architecture, conflicts can have a significant impact of the performance and/or storage requirement, and machine learned models might provide an alternative to reduce the number of conflicts. While the idea of learning models as a hash-function is not new, existing techniques do not take advantage of the underlying data distribution. For example, the various perfect hashing techniques [26] also try to avoid conflicts but the data structure used as part of the hash functions grow with the data size; a property learned models might not have (recall, the example of indexing all keys between 1 and 100M). To our knowledge it has not been explored if it is possible to learn models which yield more efficient point indexes.

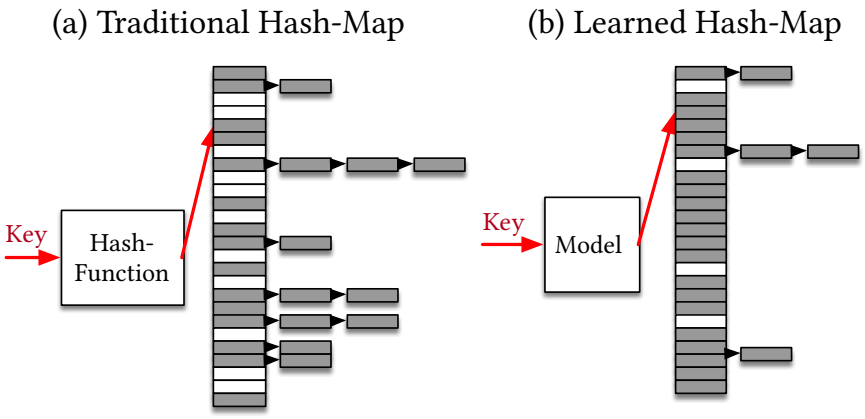

Figure 7: Traditional Hash-map vs Learned Hash-map

4.1 The Hash-Model Index

Surprisingly, learning the CDF of the key distribution is one potential way to learn a better hash function. However, in contrast to range indexes, we do not aim to store the records compactly or in strictly sorted order. Rather we can scale the CDF by the targeted size M of the Hash-map and use $h(K) = F(K) * M$, with key K as our hash-function. If the model F perfectly learned the empirical CDF of the keys, no conflicts would exist. Furthermore, the hash-function is orthogonal to the actual Hash-map architecture and can be combined with separate chaining or any other Hash-map type.

For the model, we can again leverage the recursive model architecture from the previous section. Obviously, like before, there exists a trade-off between the size of the index and performance, which is influenced by the model and dataset.

Note, that how inserts, look-ups, and conflicts are handled is dependent on the Hash-map architecture. As a result, the benefits learned hash functions provide over traditional hash functions, which map keys to a uniformly distributed space depend on two key factors: (1) How accurately the model represents the observed CDF. For example, if the data is generated by a uniform distribution, a simple linear model will be able to learn the general data distribution, but the resulting hash function will not be better than any sufficiently randomized hash function. (2) Hash map architecture: depending on the architecture, implementation details, the payload (i.e., value), the conflict resolution policy, as well as how much more memory (i.e., slots) will or can be allocated, significantly influences the performance. For example, for small keys and small or no values, traditional hash functions with Cuckoo hashing will probably work well, whereas larger payloads or distributed hash maps might benefit more from avoiding conflicts, and thus from learned hash functions.

4.2 Results

We evaluated the conflict rate of learned hash functions over the three integer data sets from the previous section. As our model hash-functions we used the 2-stage RMI models from the previous section with 100k models on the 2nd stage and without any hidden layers. As the baseline we used a simple MurmurHash3-like hash-function and compared the number of conflicts for a table with the same number of slots as records.

As can be seen in Figure 8, the learned models can reduce the number of conflicts by up to 77% over our datasets by learning the empirical CDF at a reasonable cost; the execution time is the same as the model execution time in Figure 4, around 25-40ns.

How beneficial the reduction of conflicts is given the model execution time depends on the Hash-map architecture, payload, and many other factors. For example, our experiments (see Appendix B) show that for a separate chaining Hash-map architecture with 20 Byte records learned hash functions can reduce the

	% 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%

Figure 8: Reduction of Conflicts

wasted amount of storage by up to 80% at an increase of only 13ns in latency compared to random hashing. The reason why it only increases the latency by 13ns and not 40ns is, that often fewer conflicts also yield to fewer cache misses, and thus better performance. On the other hand, for very small payloads Cuckoo-hashing with standard hash-maps probably remains the best choice. However, as we show in Appendix C, for larger payloads a chained-hashmap with learned hash function can be faster than cuckoo-hashing and/or traditional randomized hashing. Finally, we see the biggest potential for distributed settings. For example, NAM-DB [74] employs a hash function to look-up data on remote machines using RDMA. Because of the extremely high cost for every conflict (i.e., every conflict requires an additional RDMA request which is in the order of micro-seconds), the model execution time is negligible and even small reductions in the conflict rate can significantly improve the overall performance. To conclude, learned hash functions are independent of the used Hash-map architecture and depending on the Hash-map architecture their complexity may or may not pay off.

5 Existence Index

The last common index type of DBMS are existence indexes, most importantly Bloom filters, a space efficient probabilistic data structure to test whether an element is a member of a set. They are commonly used to determine if a key exists on cold storage. For example, Bigtable uses them to determine if a key is contained in an SSTable [23].

Internally, Bloom filters use a bit array of size m and k hash functions, which each map a key to one of the m array positions (see Figure9(a)). To add an element to the set, a key is fed to the k hash-functions and the bits of the returned positions are set to 1. To test if a key is a member of the set, the key is again fed into the k hash functions to receive k array positions. If any of the bits at those k positions is 0, the key is not a member of a set. In other words, a Bloom filter does guarantee that there exists *no false negatives*, but has potential *false positives*.

While Bloom filters are highly space-efficient, they can still occupy a significant amount of memory. For example for one billion records roughly ≈ 1.76 Gigabytes are needed. For a FPR of 0.01% we would require ≈ 2.23 Gigabytes. There have been several attempts to improve the efficiency of Bloom filters [52], but the general observation remains.

Yet, if there is some structure to determine what is inside versus outside the set, which can be learned, it might be possible to construct more efficient representations. Interestingly, for existence indexes for database systems, the latency and space requirements are usually quite different than what we saw before. Given the high latency to access cold storage (e.g., disk or even band), we can afford more complex models while the main objective is to minimize the space for the index and the number of false positives. We outline two potential ways to build existence indexes using lea

5.1 Learned Bloom filters

While both range and point indexes learn the distribution of keys, existence indexes need to learn a function that separates keys from everything else. Stated differently, a good hash function for a point index is one with few collisions among keys, whereas a good hash function for a Bloom filter would be one that has lots of

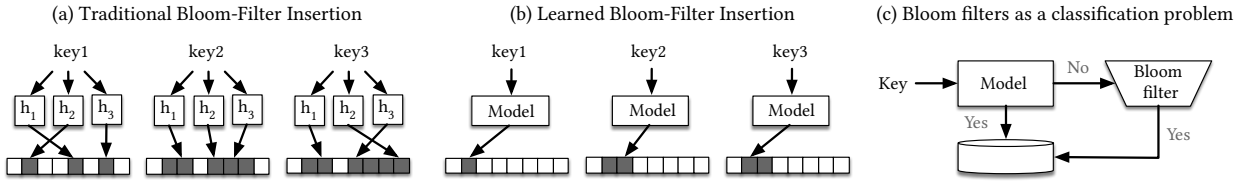

Figure 9: Bloom filters Architectures

collisions among keys and lots of collisions among non-keys, but few collisions of keys and non-keys. We consider below how to learn such a function f and how to incorporate it into an existence index.

While traditional Bloom filters guarantee a false negative rate (FNR) of zero and a specific false positive rate (FPR) for any set of queries chosen a-priori [22], we follow the notion that we want to provide a specific FPR for *realistic queries* in particular while maintaining a FNR of zero. That is, we measure the FPR over a heldout dataset of queries, as is common in evaluating ML systems [30]. While these definitions differ, we believe the assumption that we can observe the distribution of queries, e.g., from historical logs, holds in many applications, especially within databases[5].

Traditionally, existence indexes make no use of the distribution of keys nor how they differ from non-keys, but learned Bloom filters can. For example, if our database included all integers x for $0 \leq x < n$, the existence index could be computed in constant time and with almost no memory footprint by just computing $f(x) \equiv \mathbb{1}[0 \leq x < n]$.

In considering the data distribution for ML purposes, we must consider a dataset of non-keys. In this work, we consider the case where non-keys come from observable historical queries and we assume that future queries come from the same distribution as historical queries. When this assumption does not hold, one could use randomly generated keys, non-keys generated by a machine learning model [34], importance weighting to directly address covariate shift [18], or adversarial training for robustness [65]; we leave this as future work. We denote the set of keys by $\mathcal{K}$ and the set of non-keys by $\mathcal{U}$.

5.1.1 Bloom filters as a Classification Problem

One way to frame the existence index is as a binary probabilistic classification task. That is, we want to learn a model f that can predict if a query x is a key or non-key. For example, for strings we can train a recurrent neural network (RNN) or convolutional neural network (CNN) [64, 37] with $\mathcal{D} = \{(x_i, y_i = 1)|x_i \in \mathcal{K}\} \cup \{(x_i, y_i = 0)|x_i \in \mathcal{U}\}$. Because this is a binary classification task, our neural network has a sigmoid activation to produce a probability and is trained to minimize the log loss: $L = \sum_{(x,y)\in\mathcal{D}} y \log f(x) + (1 - y) \log(1 - f(x))$.

The output of $f(x)$ can be interpreted as the probability that x is a key in our database. Thus, we can turn the model into an existence index by choosing a threshold τ above which we will assume that the key exists in our database. Unlike Bloom filters, our model will likely have a non-zero FPR and FNR; in fact, as the FPR goes down, the FNR will go up. In order to preserve the no false negatives constraint of existence indexes, we create an overflow Bloom filter. That is, we consider $\mathcal{K}_\tau^- = \{x \in \mathcal{K}|f(x) < \tau\}$ to be the set of false negatives from f and create a Bloom filter for this subset of keys. We can then run our existence index as in Figure 9(c): if $f(x) \geq \tau$, the key is believed to exist; otherwise, check the overflow Bloom filter.

One question is how to set τ so that our learned Bloom filter has the desired FPR p^*. We denote the FPR of our model by $\text{FPR}_\tau \equiv \frac{\sum_{x\in\tilde{\mathcal{U}}} \mathbb{1}(f(x)>\tau)}{|\tilde{\mathcal{U}}|}$ where $\tilde{\mathcal{U}}$ is a held-out set of non-keys. We denote the FPR of our overflow Bloom filter by FPR_B. The overall FPR of our system therefore is $\text{FPR}_O =$

[5]We would like to thank Michael Mitzenmacher for valuable conversations in articulating the relationship between these definitions as well as improving the overall chapter through his insightful comments.

$\text{FPR}_\tau + (1 - \text{FPR}_\tau)\text{FPR}_B$ [53] [6]. For simplicity, we set $\text{FPR}_\tau = \text{FPR}_B = \frac{p^*}{2}$ so that $\text{FPR}_O \leq p^*$. We tune τ to achieve this FPR on $\tilde{\mathcal{U}}$.

This setup is effective in that the learned model can be fairly small relative to the size of the data. Further, because Bloom filters scale with the size of key set, the overflow Bloom filter will scale with the FNR. We will see experimentally that this combination is effective in decreasing the memory footprint of the existence index. Finally, the learned model computation can benefit from machine learning accelerators, whereas traditional Bloom filters tend to be heavily dependent on the random access latency of the memory system.

5.1.2 Bloom filters with Model-Hashes

An alternative approach to building existence indexes is to learn a hash function with the goal to maximize collisions among keys and among non-keys while minimizing collisions of keys and non-keys. Interestingly, we can use the same probabilistic classification model as before to achieve that. That is, we can create a hash function d, which maps f to a bit array of size m by scaling its output as $d = \lfloor f(x) * m \rfloor$ As such, we can use d as a hash function just like any other in a Bloom filter. This has the advantage of f being trained to map most keys to the higher range of bit positions and non-keys to the lower range of bit positions (see Figure9(b)). A more detailed explanation of the approach is given in Appendix E.

5.2 Results

In order to test this idea experimentally, we explore the application of an existence index for keeping track of blacklisted phishing URLs. We consider data from Google's transparency report as our set of keys to keep track of. This dataset consists of 1.7M unique URLs. We use a negative set that is a mixture of random (valid) URLs and whitelisted URLs that could be mistaken for phishing pages. We split our negative set randomly into train, validation and test sets. We train a character-level RNN (GRU [24], in particular) to predict which set a URL belongs to; we set τ based on the validation set and also report the FPR on the test set.

A normal Bloom filter with a desired 1% FPR requires 2.04MB. We consider a 16-dimensional GRU with a 32-dimensional embedding for each character; this model is 0.0259MB. When building a comparable learned index, we set τ for 0.5% FPR on the validation set; this gives a FNR of 55%. (The FPR on the test set is 0.4976%, validating the chosen threshold.) As described above, the size of our Bloom filter scales with the FNR. Thus, we find that our model plus the spillover Bloom filter uses 1.31MB, a 36% reduction in size. If we want to enforce an overall FPR of 0.1%, we have a FNR of 76%, which brings the total Bloom filter size down from 3.06MB to 2.59MB, a 15% reduction in memory. We observe this general relationship in Figure 10. Interestingly, we see how different size models balance the accuracy vs. memory trade-off differently.

We consider briefly the case where there is covariate shift in our query distribution that we have not addressed in the model. When using validation and test sets with only random URLs we find that we can save 60% over a Bloom filter with a FPR of 1%. When using validation and test sets with only the whitelisted URLs we find that we can save 21% over a Bloom filter with a FPR of 1%. Ultimately, the choice of negative set is application specific and covariate shift could be more directly addressed, but these experiments are intended to give intuition for how the approach adapts to different situations.

Clearly, the more accurate our model is, the better the savings in Bloom filter size. One interesting property of this is that there is no reason that our model needs to use the same features as the Bloom filter. For example, significant research has worked on using ML to predict if a webpage is a phishing page [10, 15]. Additional features like WHOIS data or IP information could be incorporated in the model, improving accuracy, decreasing Bloom filter size, and keeping the property of no false negatives.

Further, we give additional results following the approach in Section 5.1.2 in Appendix E.

[6] We again thank Michael Mitzenmacher for identifying this relation when reviewing our paper.

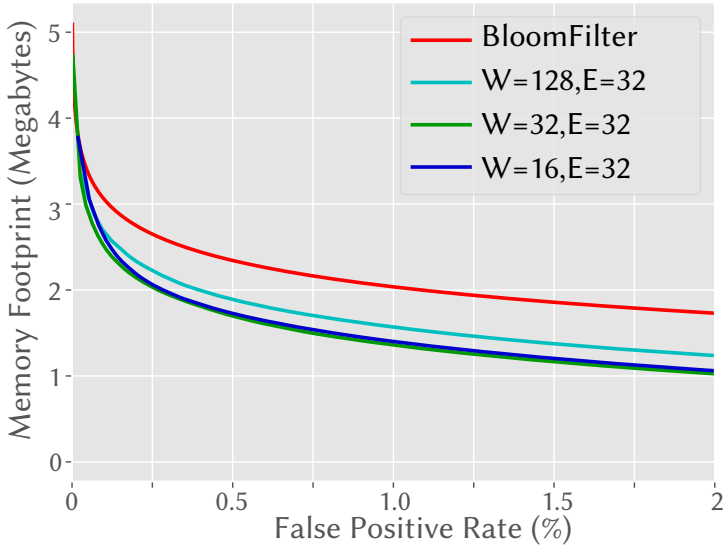

Figure 10: Learned Bloom filter improves memory footprint at a wide range of FPRs. (Here W is the RNN width and E is the embedding size for each character.)

6 Related Work

The idea of learned indexes builds upon a wide range of research in machine learning and indexing techniques. In the following, we highlight the most important related areas.

B-Trees and variants: Over the last decades a variety of different index structures have been proposed [36], such as B+-trees [17] for disk based systems and T-trees [46] or balanced/red-black trees [16, 20] for in-memory systems. As the original main-memory trees had poor cache behavior, several cache conscious B+-tree variants were proposed, such as the CSB+-tree [58]. Similarly, there has been work on making use of SIMD instructions such as FAST [44] or even taking advantage of GPUs [44, 61, 43]. Moreover, many of these (in-memory) indexes are able to reduce their storage-needs by using offsets rather than pointers between nodes. There exists also a vast array of research on index structures for text, such as tries/radix-trees [19, 45, 31], or other exotic index structures, which combine ideas from B-Trees and tries [48].

However, all of these approaches are orthogonal to the idea of learned indexes as none of them learn from the data distribution to achieve a more compact index representation or performance gains. At the same time, like with our hybrid indexes, it might be possible to more tightly integrate the existing hardware-conscious index strategies with learned models for further performance gains.

Since B+-trees consume significant memory, there has also been a lot of work in compressing indexes, such as prefix/suffix truncation, dictionary compression, key normalization [36, 33, 55], or hybrid hot/cold indexes [75]. However, we presented a radical different way to compress indexes, which—dependent on the data distribution—is able to achieve orders-of-magnitude smaller indexes and faster look-up times and potentially even changes the storage complexity class (e.g., $O(n)$ to $O(1)$). Interestingly though, some of the existing compression techniques are complimentary to our approach and could help to further improve the efficiency. For example, dictionary compression can be seen as a form of embedding (i.e., representing a string as a unique integer).

Probably most related to this paper are A-Trees [32], BF-Trees [13], and B-Tree interpolation search [35]. BF-Trees uses a B+-tree to store information about a region of the dataset, but leaf nodes are Bloom filters and do not approximate the CDF. In contrast, A-Trees use piece-wise linear functions to reduce the number of leaf-nodes in a B-Tree, and [35] proposes to use interpolation search within a B-Tree page. However, learned

indexes go much further and propose to replace the entire index structure using learned models.

Finally, sparse indexes like Hippo [73], Block Range Indexes [63], and Small Materialized Aggregates (SMAs) [54] all store information about value ranges but again do not take advantage of the underlying properties of the data distribution.

Learning Hash Functions for ANN Indexes: There has been a lot of research on learning hash functions [49, 68, 67, 59]. Most notably, there has been work on learning locality-sensitive hash (LSH) functions to build Approximate Nearest Neighborhood (ANN) indexes. For example, [66, 68, 40] explore the use of neural networks as a hash function, whereas [69] even tries to preserve the order of the multi-dimensional input space. However, the general goal of LSH is to group similar items into buckets to support nearest neighborhood queries, usually involving learning approximate similarity measures in high-dimensional input space using some variant of hamming distances. There is no direct way to adapt previous approaches to learn the fundamental data structures we consider, and it is not clear whether they can be adapted.

Perfect Hashing: Perfect hashing [26] is very related to our use of models for Hash-maps. Like our CDF models, perfect hashing tries to avoid conflicts. However, in all approaches of which we are aware, learning techniques have not been considered, and the size of the function grows with the size of the data. In contrast, learned hash functions can be independent of the size. For example, a linear model for mapping every other integer between 0 and 200M would not create any conflicts and is independent of the size of the data. In addition, perfect hashing is also not useful for B-Trees or Bloom filters.

Bloom filters: Finally, our existence indexes directly builds upon the existing work in Bloom filters [29, 11]. Yet again our work takes a different perspective on the problem by proposing a Bloom filter enhanced classification model or using models as special hash functions with a very different optimization goal than the hash-models we created for Hash-maps.

Succinct Data Structures: There exists an interesting connection between learned indexes and succinct data structures, especially rank-select dictionaries such as wavelet trees [39, 38]. However, many succinct data structures focus on H0 entropy (i.e., the number of bits that are necessary to encode each element in the index), whereas learned indexes try to learn the underlying data distribution to predict the position of each element. Thus, learned indexes might achieve a higher compression rate than H0 entropy potentially at the cost of slower operations. Furthermore, succinct data structures normally have to be carefully constructed for each use case, whereas learned indexes "automate" this process through machine learning. Yet, succinct data structures might provide a framework to further study learned indexes.

Modeling CDFs: Our models for both range and point indexes are closely tied to models of the CDF. Estimating the CDF is non-trivial and has been studied in the machine learning community [50] with a few applications such as ranking [42]. How to most effectively model the CDF is still an open question worth further investigation.

Mixture of Experts: Our RMI architecture follows a long line of research on building experts for subsets of the data [51]. With the growth of neural networks, this has become more common and demonstrated increased usefulness [62]. As we see in our setting, it nicely lets us to decouple model size and model computation, enabling more complex models that are not more expensive to execute.

7 Conclusion and Future Work

We have shown that learned indexes can provide significant benefits by utilizing the distribution of data being indexed. This opens the door to many interesting research questions.

Other ML Models: While our focus was on linear models and neural nets with mixture of experts, there exist many other ML model types and ways to combine them with traditional data structures, which are worth exploring.

Multi-Dimensional Indexes: Arguably the most exciting research direction for the idea of learned indexes is to extend them to multi-dimensional indexes. Models, especially NNs, are extremely good at capturing complex high-dimensional relationships. Ideally, this model would be able to estimate the position of all records filtered by any combination of attributes.

Beyond Indexing: Learned Algorithms Maybe surprisingly, a CDF model has also the potential to speed-up sorting and joins, not just indexes. For instance, the basic idea to speed-up sorting is to use an existing CDF model F to put the records roughly in sorted order and then correct the nearly perfectly sorted data, for example, with insertion sort.

GPU/TPUs Finally, as mentioned several times throughout this paper, GPU/TPUs will make the idea of learned indexes even more valuable. At the same time, GPU/TPUs also have their own challenges, most importantly the high invocation latency. While it is reasonable to assume that probably all learned indexes will fit on the GPU/TPU because of the exceptional compression ratio as shown before, it still requires 2-3 micro-seconds to invoke any operation on them. At the same time, the integration of machine learning accelerators with the CPU is getting better [6, 4] and with techniques like batching requests the cost of invocation can be amortized, so that we do not believe the invocation latency is a real obstacle.

In summary, we have demonstrated that machine learned models have the potential to provide significant benefits over state-of-the-art indexes, and we believe this is a fruitful direction for future research.

Acknowledgements: We would like to thank Michael Mitzenmacher, Chris Olston, Jonathan Bischof and many others at Google for their helpful feedback during the preparation of this paper.

References

[1] Database architects blog: The case for b-tree index structures. http://databasearchitects.blogspot.de/2017/12/the-case-for-b-tree-index-structures.html.

[2] Google's sparsehash documentation. https://github.com/sparsehash/sparsehash/blob/master/src/sparsehash/sparse_hash_map.

[3] An in-depth look at google's first tensor processing unit (tpu). https://cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu.

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

[5] 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.

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

[7] Stanford DAWN cuckoo hashing. https://github.com/stanford-futuredata/index-baselines.

[8] Trying to speed up binary search. http://databasearchitects.blogspot.com/2015/09/trying-to-speed-up-binary-search.html.

[9] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[10] S. Abu-Nimeh, D. Nappa, X. Wang, and S. Nair. A comparison of machine learning techniques for phishing detection. In *eCrime*, pages 60–69, 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. Armbrust, A. Fox, D. A. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh. SCADS: scale-independent storage for social computing applications. In *CIDR*, 2009.

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

[14] Performance comparison: linear search vs binary search. https://dirtyhandscoding.wordpress.com/2017/08/25/performance-comparison-linear-search-vs-binary-search/

[15] R. B. Basnet, S. Mukkamala, and A. H. Sung. Detection of phishing attacks: A machine learning approach. *Soft Computing Applications in Industry*, 226:373–383, 2008.

[16] R. Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Inf.*, 1(4):290–306, Dec. 1972.

[17] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *SIGFIDET (Now SIGMOD)*, pages 107–141, 1970.

[18] S. Bickel, M. Brückner, and T. Scheffer. Discriminative learning under covariate shift. *Journal of Machine Learning Research*, 10(Sep):2137–2155, 2009.

[19] M. Böhm, B. Schlegel, P. B. Volk, U. Fischer, D. Habich, and W. Lehner. Efficient in-memory indexing with generalized prefix trees. In *BTW*, pages 227–246, 2011.

[20] J. Boyar and K. S. Larsen. Efficient rebalancing of chromatic search trees. *Journal of Computer and System Sciences*, 49(3):667 – 682, 1994. 30th IEEE Conference on Foundations of Computer Science.

[21] M. Brantner, D. Florescu, D. A. Graf, D. Kossmann, and T. Kraska. Building a database on S3. In *SIGMOD*, pages 251–264, 2008.

[22] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet mathematics*, 1(4):485–509, 2004.

[23] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data (awarded best paper!). In *OSDI*, pages 205–218, 2006.

[24] K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *EMNLP*, pages 1724–1734, 2014.

[25] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Çetintemel, and S. Zdonik. An architecture for compiling udf-centric workflows. *PVLDB*, 8(12):1466–1477, 2015.

[26] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auF der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

[27] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[28] A. Dvoretzky, J. Kiefer, and J. Wolfowitz. Asymptotic minimax character of the sample distribution function and of the classical multinomial estimator. *The Annals of Mathematical Statistics*, pages 642–669, 1956.

[29] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *CoNEXT*, pages 75–88, 2014.

[30] T. Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[31] E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, Sept. 1960.

[32] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska. A-tree: A bounded approximate index structure. *CoRR*, abs/1801.10207, 2018.

[33] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing Relations and Indexes. In *ICDE*, pages 370–379, 1998.

[34] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *NIPS*, pages 2672–2680, 2014.

[35] G. Graefe. B-tree indexes, interpolation search, and skew. In *DaMoN*, 2006.

[36] G. Graefe and P. A. Larson. B-tree indexes and CPU caches. In *ICDE*, pages 349–358, 2001.

[37] A. Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[38] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *SODA*, pages 841–850. Society for Industrial and Applied Mathematics, 2003.

[39] R. Grossi and G. Ottaviano. The wavelet trie: Maintaining an indexed sequence of strings in compressed space. In *PODS*, pages 203–214, 2012.

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

[41] M. Gupta, A. Cotter, J. Pfeifer, K. Voevodski, K. Canini, A. Mangylov, W. Moczydlowski, and A. Van Esbroeck. Monotonic calibrated interpolated look-up tables. *The Journal of Machine Learning Research*, 17(1):3790–3836, 2016.

[42] 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.

[43] K. Kaczmarski. *B + -Tree Optimized for GPGPU*. 2012.

[44] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: Fast architecture sensitive tree search on modern cpus and gpus. In *SIGMOD*, pages 339–350, 2010.

[45] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner. Kiss-tree: Smart latch-free in-memory indexing on modern architectures. In *DaMoN*, pages 16–23, 2012.

[46] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *VLDB*, pages 294–303, 1986.

[47] V. Leis. FAST source. `http://www-db.in.tum.de/leis/index/fast.cpp`.

[48] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *ICDE*, pages 38–49, 2013.

[49] W. Litwin. Readings in database systems. chapter Linear Hashing: A New Tool for File and Table Addressing., pages 570–581. Morgan Kaufmann Publishers Inc., 1988.

[50] M. Magdon-Ismail and A. F. Atiya. Neural networks for density estimation. In M. J. Kearns, S. A. Solla, and D. A. Cohn, editors, *NIPS*, pages 522–528. MIT Press, 1999.

[51] D. J. Miller and H. S. Uyar. A mixture of experts classifier with learning based on both labelled and unlabelled data. In *NIPS*, pages 571–577, 1996.

[52] M. Mitzenmacher. Compressed bloom filters. In *PODC*, pages 144–150, 2001.

[53] M. Mitzenmacher. A model for learned bloom filters and related structures. *arXiv preprint arXiv:1802.00884*, 2018.

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

[55] T. Neumann and G. Weikum. RDF-3X: A RISC-style Engine for RDF. *Proc. VLDB Endow.*, pages 647–659, 2008.

[56] OpenStreetMap database ©OpenStreetMap contributors. `https://aws.amazon.com/public-datasets/osm`.

[57] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[58] J. Rao and K. A. Ross. Making b+- trees cache conscious in main memory. In *SIGMOD*, pages 475–486, 2000.

[59] 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.

[60] D. G. Severance and G. M. Lohman. Differential files: Their application to the maintenance of large data bases. In *SIGMOD*, pages 43–43, 1976.

[61] A. Shahvarani and H.-A. Jacobsen. A hybrid b+-tree as solution for in-memory indexing on cpu-gpu heterogeneous computing platforms. In *SIGMOD*, pages 1523–1538, 2016.

[62] 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.

[63] M. Stonebraker and L. A. Rowe. The Design of POSTGRES. In *SIGMOD*, pages 340–355, 1986.

[64] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *NIPS*, pages 3104–3112, 2014.

[65] F. Tramèr, A. Kurakin, N. Papernot, D. Boneh, and P. McDaniel. Ensemble adversarial training: Attacks and defenses. *arXiv preprint arXiv:1705.07204*, 2017.

[66] M. Turcanik and M. Javurek. Hash function generation by neural network. In *NTSP*, pages 1–5, Oct 2016.

[67] J. Wang, W. Liu, S. Kumar, and S. F. Chang. Learning to hash for indexing big data;a survey. *Proceedings of the IEEE*, 104(1):34–57, Jan 2016.

[68] J. Wang, H. T. Shen, J. Song, and J. Ji. Hashing for similarity search: A survey. *CoRR*, abs/1408.2927, 2014.

[69] J. Wang, J. Wang, N. Yu, and S. Li. Order preserving hashing for approximate nearest neighbor search. In *MM*, pages 133–142, 2013.

[70] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[71] S. You, D. Ding, K. Canini, J. Pfeifer, and M. Gupta. Deep lattice networks and partial monotonic functions. In *NIPS*, pages 2985–2993, 2017.

[72] Y. You, Z. Zhang, C. Hsieh, J. Demmel, and K. Keutzer. Imagenet training in minutes. *CoRR, abs/1709.05011*, 2017.

[73] 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.

[74] E. Zamanian, C. Binnig, T. Kraska, and T. Harris. The end of a myth: Distributed transaction can scale. *PVLDB*, 10(6):685–696, 2017.

[75] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen. Reducing the storage overhead of main-memory OLTP databases with hybrid indexes. In *SIGMOD*, pages 1567–1581, 2016.

A Theoretical Analysis of Scaling Learned Range Indexes

One advantage of framing learned range indexes as modeling the cumulative distribution function (CDF) of the data is that we can build on the long research literature on modeling the CDF. Significant research has studied the relationship between a theoretical CDF $F(x)$ and the empirical CDF of data sampled from $F(x)$. We consider the case where we have sampled i.i.d. N datapoints, $\mathcal{Y}$, from some distribution, and we will use $\hat{F}_N(x)$ to denote the empirical cumulative distribution function:

$$\hat{F}_N(x) = \frac{\sum_{y \in \mathcal{Y}} \mathbf{1}_{y \leq x}}{N}. \tag{2}$$

One theoretical question about learned indexes is: how well do they scale with the size of the data N? In our setting, we learn a model $F(x)$ to approximate the distribution of our data $\hat{F}_N(x)$. Here, we assume we know the distribution $F(x)$ that generated the data and analyze the error inherent in the data being sampled from that distribution[7]. That is, we consider the error between the distribution of data $\hat{F}_N(x)$ and our model of the distribution $F(x)$. Because $\hat{F}_N(x)$ is a binomial random variable with mean $F(x)$, we find that the expected squared error between our data and our model is given by

$$\mathbf{E}\left[\left(F(x) - \hat{F}_N(x)\right)^2\right] = \frac{F(x)(1 - F(x))}{N}. \tag{3}$$

In our application the look-up time scales with the average error in the number of positions in the sorted data; that is, we are concerned with the error between our model $NF(x)$ and the key position $N\hat{F}_N(x)$. With some minor manipulation of Eq. (3), we find that the average error in the predicted positions grows at a rate of $O(\sqrt{N})$. Note that this sub-linear scaling in error for a constant-sized model is an improvement over the linear scaling achieved by a constant-sized B-Tree. This provides preliminary understanding of the scalability of our approach and demonstrates how framing indexing as learning the CDF lends itself well to theoretical analysis.

B Separated Chaining Hash-map

We evaluated the potential of learned hash functions using a separate chaining Hash-map; records are stored directly within an array and only in the case of a conflict is the record attached to the linked-list. That is without a conflict there is at most one cache miss. Only in the case that several keys map to the same position, additional cache-misses might occur. We choose that design as it leads to the best look-up performance even for larger payloads. For example, we also tested a commercial-grade dense Hash-map with a bucket-based in-place overflow (i.e., the Hash-map is divided into buckets to minimize overhead and uses in-place overflow if a bucket is full [2]). While it is possible to achieve a lower footprint using this technique, we found that it is also twice as slow as the separate chaining approach. Furthermore, at 80% or more memory utilization the dense Hash-maps degrade further in performance. Of course many further (orthogonal) optimizations are possible and by no means do we claim that this is the most memory or CPU efficient implementation of a Hash-map. Rather we aim to demonstrate the general potential of learned hash functions.

As the baseline for this experiment we used our Hash-map implementation with a MurmurHash3-like hash-function. As the data we used the three integer datasets from Section 3.7 and as the model-based Hash-map the 2-stage RMI model with 100k models on the 2nd stage and no hidden layers from the same section. For all experiments we varied the number of available slots from 75% to 125% of the data. That is, with 75% there are 25% less slots in the Hash-map than data records. Forcing less slots than the data

[7]Learning $F(x)$ can improve or worsen the error, but we take this as a reasonable assumption for some applications, such as data keyed by a random hash.

Dataset	Slots	Hash Type	Time (ns)	Empty Slots	Space
Map	75%	Model Hash	67	0.18GB	0.21x
		Random Has	52	0.84GB	
	100%	Model Hash	53	0.35GB	0.22x
		Random Has	48	1.58GB	
	125%	Model Hash	64	1.47GB	0.60x
		Random Has	49	2.43GB	
Web	75%	Model Hash	78	0.64GB	0.77x
		Random Has	53	0.83GB	
	100%	Model Hash	63	1.09GB	0.70x
		Random Has	50	1.56GB	
	125%	Model Hash	77	2.20GB	0.91x
		Random Has	50	2.41GB	
Log Normal	75%	Model Hash	79	0.63GB	0.79x
		Random Has	52	0.80GB	
	100%	Model Hash	66	1.10GB	0.73x
		Random Has	46	1.50GB	
	125%	Model Hash	77	2.16GB	0.94x
		Random Has	46	2.31GB	

Figure 11: Model vs Random Hash-map

size, minimizes the empty slots within the Hash-map at the expense of longer linked lists. However, for Hash-maps we store the full records, which consist of a 64bit key, 64bit payload, and a 32bit meta-data field for delete flags, version nb, etc. (so a record has a fixed length of 20 Bytes); note that our chained hash-map adds another 32bit pointer, making it a 24Byte slot.

The results are shown in Figure 11, listing the average look-up time, the number of empty slots in GB and the space improvement as a factor of using a randomized hash function. Note, that in contrast to the B-Tree experiments, we *do include the data size*. The main reason is that in order to enable 1 cache-miss look-ups, the data itself has to be included in the Hash-map, whereas in the previous section we only counted the extra index overhead excluding the sorted array itself.

As can be seen in Figure 11, the index with the model hash function overall has similar performance while utilizing the memory better. For example, for the map dataset the model hash function only "wastes" 0.18GB in slots, an almost 80% reduction compared to using a random hash function. Obviously, the moment we increase the Hash-map in size to have 25% more slots, the savings are not as large, as the Hash-map is also able to better spread out the keys. Surprisingly if we decrease the space to 75% of the number of keys, the learned Hash-map still has an advantage because of the still prevalent birthday paradox.

C Hash-Map Comparison Against Alternative Baselines

In addition to the separate chaining Hash-map architecture, we also compared learned point indexes against four alternative Hash-map architectures and configurations:

AVX Cuckoo Hash-map: We used an AVX optimized Cuckoo Hash-map from [7].

Commercial Cuckoo Hash-map: The implementation of [7] is highly tuned, but does not handle all corner cases. We therefore also compared against a commercially used Cuckoo Hash-map.

In-place chained Hash-map with learned hash functions: One significant downside of separate chaining is that it requires additional memory for the linked list. As an alternative, we implemented a chained Hash-map, which uses a two pass algorithm: in the first pass, the learned hash function is used to put items into slots. If a slot is already taken, the item is skipped. Afterwards we use a separate chaining approach for every skipped item except that we use the remaining free slots with offsets as pointers for them. As a result,

the utilization can be 100% (recall, we do not consider inserts) and the quality of the learned hash function can only make an impact on the performance not the size: the fewer conflicts, the fewer cache misses. We used a simple single stage multi-variate model as the learned hash function and implemented the Hash-map including the model outside of our benchmarking framework to ensure a fair comparison.

Type	Time (ns)	Utilization
AVX Cuckoo, 32-bit value	31ns	99%
AVX Cuckoo, 20 Byte record	43ns	99%
Comm. Cuckoo, 20Byte record	90ns	95%
In-place chained Hash-map with learned hash functions, record	35ns	100%

Table 1: Hash-map alternative baselines

Like in Section B our records are 20 Bytes large and consist of a 64bit key, 64bit payload, and a 32bit meta-data field as commonly found in real applications (e.g., for delete flags, version numbers, etc.). For all Hash-map architectures we tried to maximize utilization and used records, except for the AVX Cuckoo Hash-map where we also measured the performance for 32bit values. As the dataset we used the log-normal data and the same hardware as before. The results are shown in Table 1.

The results for the AVX cuckoo Hash-map show that the payload has a significant impact on the performance. Going from 8 Byte to 20 Byte decreases the performance by almost 40%. Furthermore, the commercial implementation which handles all corner cases but is not very AVX optimized slows down the lookup by another factor of 2. In contrast, our learned hash functions with in-place chaining can provide better lookup performance than even the cuckoo Hash-map for our records. The main take-aways from this experiment is that learned hash functions can be used with different Hash-map architectures and that the benefits and disadvantages highly depend on the implementation, data and workload.

D Future Directions for Learned B-Trees

In the main part of the paper, we have focused on index-structures for read-only, in-memory database systems. Here we outline how the idea of learned index structures could be extended in the future.

D.1 Inserts and Updates

On first sight, inserts seem to be the Achilles heel of learned indexes because of the potentially high cost for learning models, but yet again learned indexes might have a significant advantage for certain workloads. In general we can distinguish between two types of inserts: (1) *appends* and (2) *inserts in the middle* like updating a secondary index on the customer-id over an order table.

Let's for the moment focus on the first case: appends. For example, it is reasonable to assume that for an index over the timestamps of web-logs, like in our previous experiments, most if not all inserts will be appends with increasing timestamps. Now, let us further assume that our model generalizes and is able to learn the patterns, which also hold for the future data. As a result, updating the index structure becomes an $O(1)$ operation; it is a simple append and no change of the model itself is needed, whereas a B-Tree requires $O(\log n)$ operations to keep the B-Tree balance. A similar argument can also be made for inserts in the middle, however, those might require to move data or reserve space within the data, so that the new items can be put into the right place.

Obviously, this observation also raises several questions. First, there seems to be an interesting trade-off in the generalizability of the model and the "last mile" performance; the better the "last mile" prediction, arguably, the more the model is overfitting and less able to generalize to new data items.

Second, what happens if the distribution changes? Can it be detected, and is it possible to provide similar strong guarantees as B-Trees which always guarantee $O(logn)$ look-up and insertion costs? While answering this question goes beyond the scope of this paper, we believe that it is possible for certain models to achieve it. More importantly though, machine learning offers new ways to adapt the models to changes in the data distribution, such as online learning, which might be more effective than traditional B-Tree balancing techniques. Exploring them also remains future work.

Finally, it should be pointed out that there always exists a much simpler alternative to handling inserts by building a delta-index [60]. All inserts are kept in buffer and from time to time merged with a potential retraining of the model. This approach is already widely used, for example in Bigtable [23] and many other systems, and was recently explored in [32] for learned indexes.

D.2 Paging

Throughout this section we assumed that the data, either the actual records or the `<key,pointer>` pairs, are stored in one continuous block. However, especially for indexes over data stored on disk, it is quite common to partition the data into larger pages that are stored in separate regions on disk. To that end, our observation that a model learns the CDF no longer holds true as $pos = \Pr(X < \text{Key}) * N$ is violated. In the following we outline several options to overcome this issue:

Leveraging the RMI structure: The RMI structure already partitions the space into regions. With small modifications to the learning process, we can minimize how much models overlap in the regions they cover. Furthermore, it might be possible to duplicate any records which might be accessed by more than one model.

Another option is to have an additional translation table in the form of `<first_key, disk-position>`. With the translation table the rest of the index structure remains the same. However, this idea will work best if the disk pages are very large. At the same time it is possible to use the predicted position with the min- and max-error to reduce the number of bytes which have to be read from a large page, so that the impact of the page size might be negligible.

With more complex models, it might actually be possible to learn the actual pointers of the pages. Especially if a file-system is used to determine the page on disk with a systematic numbering of the blocks on disk (e.g., `block1,...,block100`) the learning process can remain the same.

Obviously, more investigation is required to better understand the impact of learned indexes for disk-based systems. At the same time the significant space savings as well as speed benefits make it a very interesting avenue for future work.

E Further Bloom filter Results

In Section 5.1.2, we propose an alternative approach to a learned Bloom filter where the classifier output is discretized and used as an additional hash function in the traditional Bloom filter setup. Preliminary results demonstrate that this approach in some cases outperforms the results listed in Section 5.2, but as the results depend on the discretization scheme, further analysis is worthwhile. We describe below these additional experiments.

As before, we assume we have a model model $f(x) \rightarrow [0, 1]$ that maps keys to the range $[0, 1]$. In this case, we allocate m bits for a bitmap M where we set $M[\lfloor mf(x) \rfloor] = 1$ for all inserted keys $x \in \mathcal{K}$. We can then observe the FPR by observing what percentage of non-keys in the validation set map to a location in the bitmap with a value of 1, i.e. $\text{FPR}_m \equiv \frac{\sum_{x \in \tilde{\mathcal{U}}} M[\lfloor f(x)m \rfloor]}{|\tilde{\mathcal{U}}|}$. In addition, we have a traditional Bloom filter with false positive rate FPR_B. We say that a query q is predicted to be a key if $M[\lfloor f(q)m \rfloor] = 1$ and the Bloom filter also returns that it is a key. As such, the overall FPR of the system is $\text{FPR}_m \times \text{FPR}_B$; we can

determine the size of the traditional Bloom filter based on it's false positive rate $\mathrm{FPR}_B = \frac{p^*}{\mathrm{FPR}_m}$ where p^* is the desired FPR for the whole system.

As in Section 5.2, we test our learned Bloom filter on data from Google's transparency report. We use the same character RNN trained with a 16-dimensional width and 32-dimensional character embeddings. Scanning over different values for m, we can observe the total size of the model, bitmap for the learned Bloom filter, and the traditional Bloom filter. For a desired total FPR $p^* = 0.1\%$, we find that setting $m = 1000000$ gives a total size of 2.21MB, a 27.4% reduction in memory, compared to the 15% reduction following the approach in Section 5.1.1 and reported in Section 5.2. For a desired total FPR $p^* = 1\%$ we get a total size of 1.19MB, a 41% reduction in memory, compared to the 36% reduction reported in Section 5.2.

These results are a significant improvement over those shown in Section 5.2. However, typical measures of accuracy or calibration do not match this discretization procedure, and as such further analysis would be valuable to understand how well model accuracy aligns with it's suitability as a hash function.