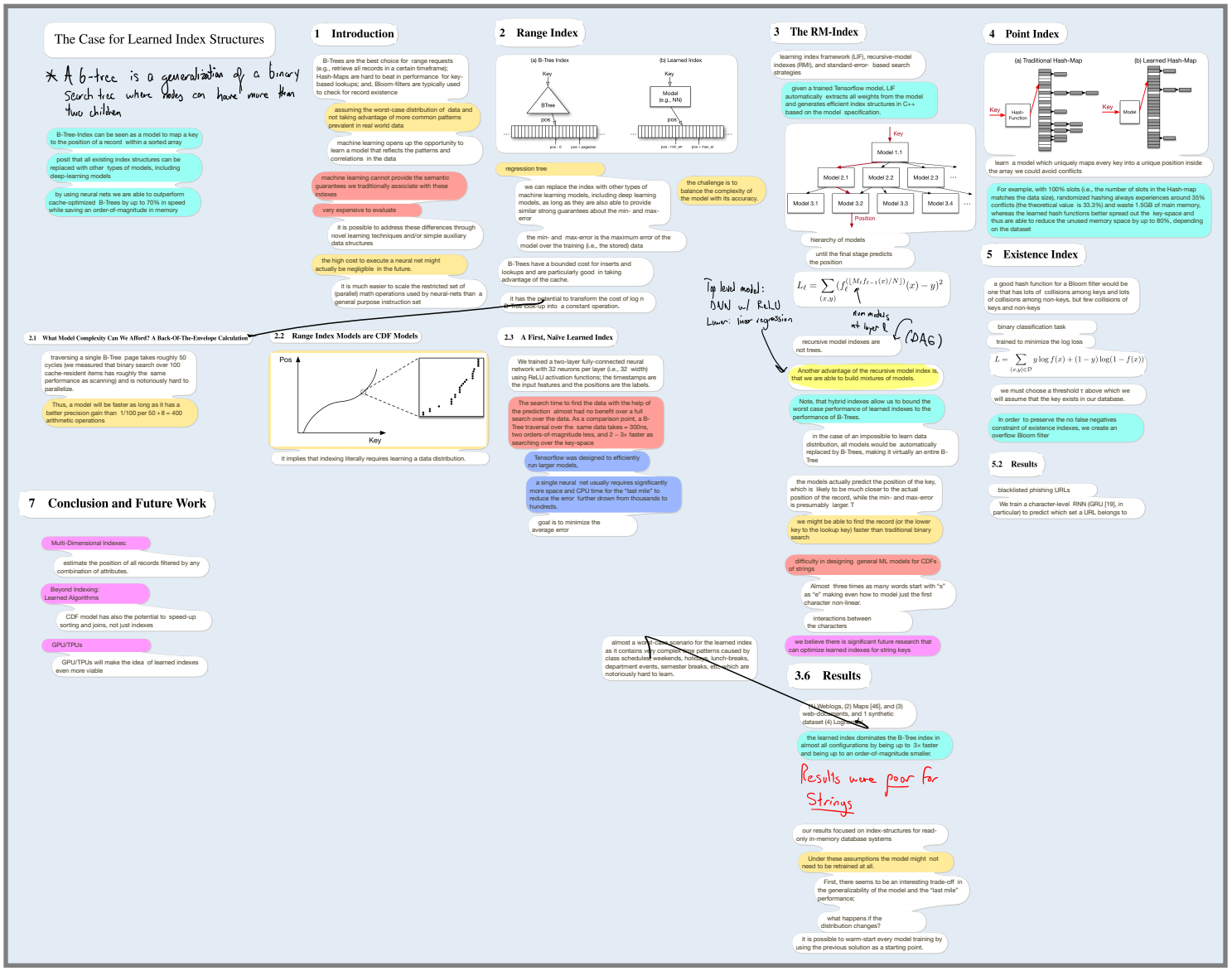
Workspace for 'The Case for Learned Index Structures - Kraska et al. 2017'

Page 1 (row 1, column 1)



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 pattern. For example, B-Trees are the best choice for range requests (e.g., retrieve all records in a certain timeframe); Hash-Maps are hard to beat in performance for key-based lookups; and, Bloom-filters are typically used to check for record existence. Because of the importance of indexes for database systems and many other applications, they have been extensively tuned over the past decades to be more memory, cache and/or CPU efficient [28, 48, 22, 11].

Yet, all of those indexes remain general purpose data structures, assuming the worst-case distribution of data and not taking advantage of more common patterns prevalent in real world data. For example, if the goal would be to build a highly tuned system to store and query fixed-length records with 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 from $O(n)$ to $O(1)$. Maybe

*Work done while author was affiliated with Google.

surprisingly, the same optimizations are still possible for other data patterns. In other words, knowing the exact data distributions enables highly optimizing almost any index the database system uses.

Of course, in most real-world use cases the data does 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 opens up the opportunity to learn a model that reflects the patterns and correlations in the data and thus 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 replace traditional index structures from B-Trees to Bloom-Filters. This may seem counter-intuitive because machine learning cannot provide the semantic guarantees we traditionally associate with these indexes, and because the most powerful machine learning models, neural networks, are traditionally thought of as being very expensive to evaluate. 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 huge 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 models, like neural networks. 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. 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 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 CPU essentially is dead [5]. By replacing branch-heavy index structures with neural networks, databases can benefit from these hardware trends.

It is important to note that we do not argue to completely replace traditional index structures with learned index structures. Rather, we outline a novel approach to build indexes, which complements existing work and, arguably, opens up an entirely new research direction for a decades-old field. While our focus is on read-only analytical workloads, we also sketch how the idea could be extended to speed-up indexes for write-heavy workloads. Furthermore, we briefly outline how the same principle can be used to replace other components and operations of a database system, including sorting and joins. If successful, this could lead to a radical departure from the way database systems are currently developed.

The remainder of this paper is outlined as follows: In the next Section we introduce the general idea of learned indexes using B-Trees as an example. In Section 4 we extend this idea to hash-indexes and in Section 5 to Bloom-Filters. All sections contain a separate evaluation and list open challenges. Finally in Section 6 we discuss related work and conclude in Section 7.

2 Range Index

Index structures are already models, because they "predict" the location of a value given a key. 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 lookup key into a position

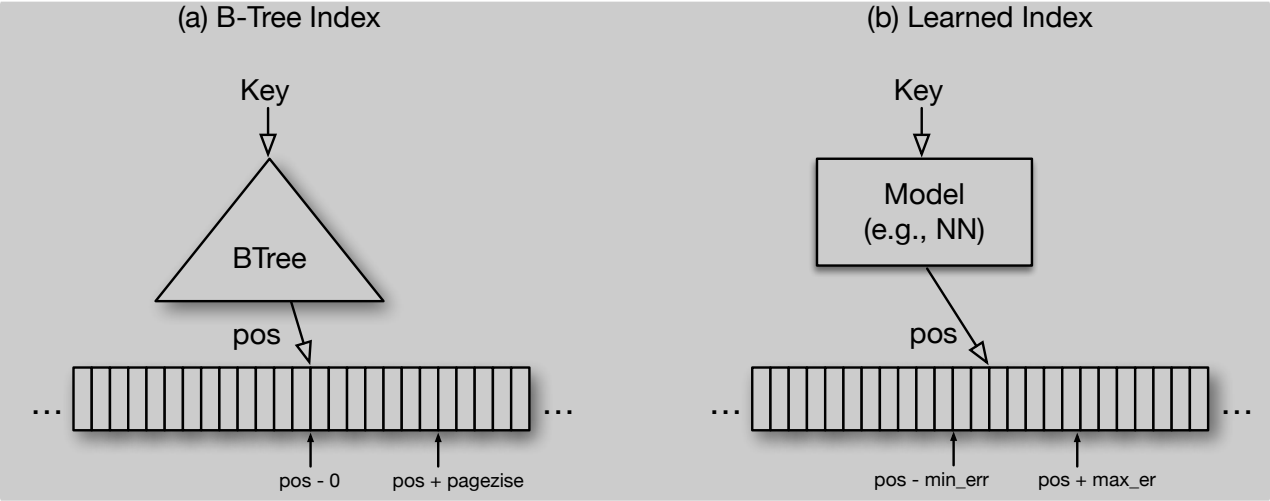


Figure 1: Why B-Trees are models

inside the sorted array of records with the guarantee that the key of the record at the position is equal or higher than the lookup key. Note that the data has to be sorted to allow for range requests. Also note that this same general concept applies to secondary indexes where the bottom layer would be the list of `<key,pointer>` pairs with the key being the value of the indexed attribute and the pointer a reference to the record.

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.[1] This 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, 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) and the guarantee that the key can be found in that region if it exists. Consequently, we can replace the index with other types of machine learning models, including deep learning models, as long as they are also able to provide similar strong guarantees about the min- and max-error.

At first sight it may be hard to provide the same error guarantees with other types of ML models, but it is actually surprisingly simple. The B-Tree only provides this guarantee over the stored data, not for all possible data. 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. This significantly simplifies the problem: the min- and max-error is the maximum error of the model over the training (i.e., the stored) data. That is, the only thing we need to do is to execute the model for every key and remember the worst over- and under-prediction of a position. Given a key, the model makes a prediction about the position where to find the data; if the key exists, it is guaranteed to be in the range of the prediction defined by the min- and max-error. 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 lookups and are particularly good in taking advantage of the cache. Also, B-Trees can map keys to pages which are not continuously mapped to memory or disk. Furthermore, if a lookup key does not exist in the set, certain models might return positions outside the min-/max-error range if they are not monotonically increasing models. All of these are interesting challenges/research questions and are explained in more detail together with potential solutions throughout this section.

[1]Here we assume logical paging over the sorted array, not physical pages which are located in different memory regions. The latter is particular needed for inserts and/or disk-based systems; we will address real paging later in the paper. Also we assume fixed-length records and with points to overflow regions for variable-length records.

At the same time, using other types of models, especially deep learning models, as indexes can provide tremendous benefits. Most importantly, it has the potential to transform the cost of $\log n$ B-Tree look-up into a constant operation. For example, assume a data set 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 consist of a single multiplication and addition, can perfectly predict the position of any key, 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. Obviously the challenge is to balance the complexity of the model with its accuracy.

2.1 What Model Complexity Can We Afford? A Back-Of-The-Envelope Calculation

In order 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 takes roughly 50 cycles (we measured that binary search over 100 cache-resident items has roughly the same performance as scanning) and is notoriously hard to parallelize.[2] 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, 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) [7]. Assuming that the entire learned index fits into the GPU's memory (we show in Section 3.6 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, roughly 2 micro-seconds or thousands of cycles, 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 the main reason to adopt learned indexes in practice, in this paper we focus on the more limited CPUs to better study the implications of replacing/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 which 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 timeframe)

[2]There exist SIMD optimized index structures such as FAST [36], but they can only transform control dependencies (i.e., if-statements) to memory dependencies (i.e., the memory location to fetch next, like the next node, depends on an outcome of a previous operation). These are often significantly slower than multiplications with simple data dependencies and as our experiments show SIMD optimized index structures, like FAST, are not necessarily faster).

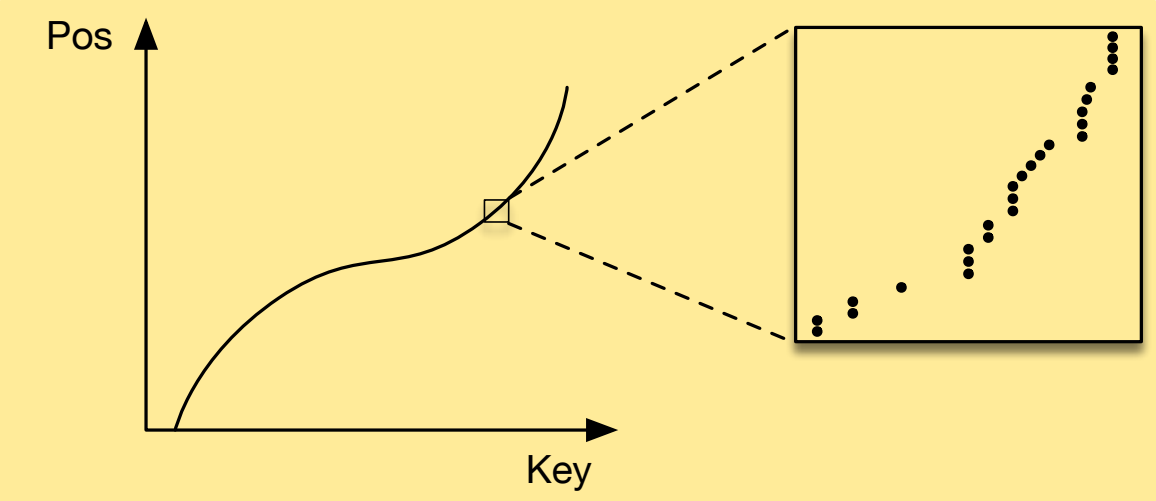


Figure 2: Indexes as CDFs

can be efficiently retrieved. This leads to an interesting observation: a model which 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 lookup 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 data set 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.

2.3 A First, Naïve Learned Index

To better understand the technical requirements to replace traditional 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 (i.e., 32 width) using ReLU activation functions; the timestamps are the input features and the positions are the labels.

Afterwards we measured the lookup 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 $\approx$ 1250 predictions per second, i.e., it takes $\approx$ 80,000 nano-seconds (ns) to execute the model with Tensorflow, without even the search time. The search time to find the data with the help of the prediction almost had no benefit over a full search over the data. As a comparison point, a B-Tree traversal over the same data takes $\approx 300ns$, two orders-of-magnitude less, and $2-3\times$ faster as searching over the key-space. The reasons for it are manifold:

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-statement. In contrast, other models can be significantly more efficient to approximate the general shape of a CDF, but have problems to be 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. Many data sets have exactly this behavior: from the top the data distribution appears very smooth, whereas as more is zoomed in the harder it is to approximate the CDF because of the "randomness" on the individual level. 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 data set to a region of thousands, a single neural net usually requires significantly more space and CPU time for the "last mile" to reduce the error further drown from thousands to hundreds.

3. The typical ML optimization goal is to minimize the average error. However, for indexing, where we not only need the best guess where the item might be but also to actually find it, the min- and max-error as discussed earlier are more important.

4. B-Trees are extremely cache-efficient as they keep the top nodes always in cache and access other pages if needed. However, other models are not as cache and operation efficient. For example, standard neural nets require all weights to compute a prediction, which has a high cost in the number of multiplications and weights, which have to brought in from memory.

3 The RM-Index

In order to overcome the challenges and explore the potential of models as index replacements or enrichments, we developed the **learning index framework** (LIF), recursive-model indexes (RMI), and standard-error-based search strategies. We mainly focus on simple, fully-connected neural nets simply because of their simplicity, but many other types of models are possible.

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. While Tensorflow with XLA already supports code compilation, its focus is mainly on large scale computations, where the execution of a model is on the scale of micro- or milli-seconds. In contrast, our code-generation focuses on small models and thus, has to remove all unnecessary overhead and instrumentation that Tensorflow has to manage the larger models. Here we leverage ideas from [21], which already showed how to avoid unnecessary overhead from the Spark-runtime. As a result, we are able to execute simple models on the order of 30 nano-seconds.

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 last-mile accuracy. For example, reducing the min-/max-error in the order of hundreds from 100M records using a single model is very hard. At the same time, reducing the error to 10k from 100M, e.g., a

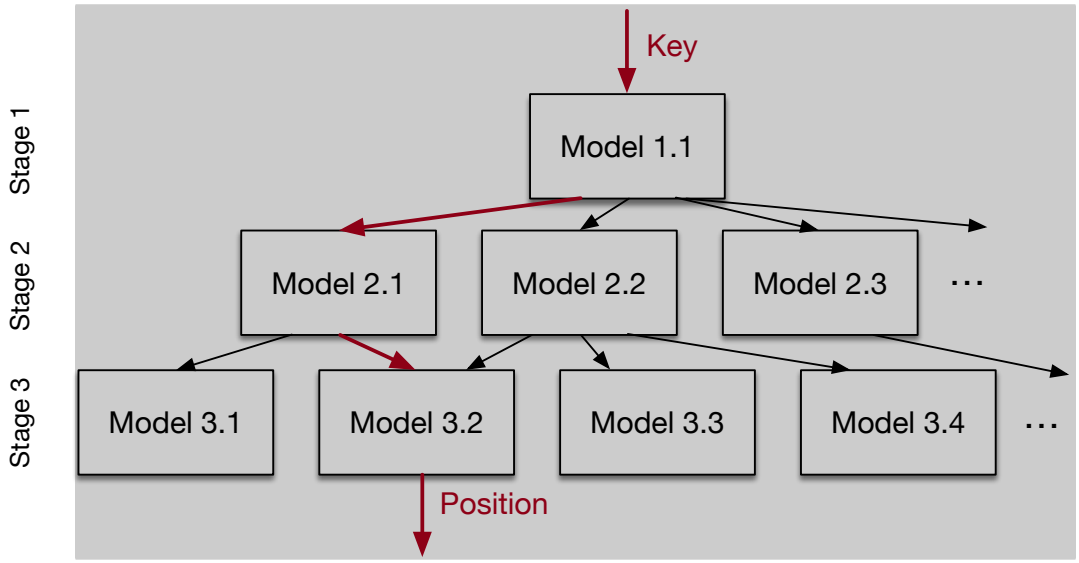


Figure 3: Staged models

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 [51], 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:

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

Note, we use here the notation here of $f_{\ell-1}(x)$ recursively executing $f_{\ell-1}(x) = f_{\ell-1}^{(\lfloor M_{\ell-1} f_{\ell-2}(x)/N \rfloor)}(x)$. Therefore, 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, it is important to note, that recursive model indexes are **not 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).[3] 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 [51]).

This model architecture has several benefits: (1) it leverages the fact, that it is easy to learn the overall shape of the data distribution. (2) The architecture effectively divides the space into smaller sub-ranges like a B-Tree/decision tree to make it easier to achieve the required "last mile" accuracy with a fewer number of operations. (3) There is no search process required in-between the stages. For example, the output y of *Model 1.1* is an offset, which can be directly used to pick the model in the next stage. This not only reduces

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

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 only focused 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). Given an index configuration, which specifies the number of stages and the number of model per stage as an array of sizes, the end-to-end training for hybrid indexes is done as 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 = f(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 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. In addition, one might wonder how to set the various parameters of our hybrid end-to-end training including the number and width of stages, neural net configuration (i.e., number of hidden layers and width), and the threshold to replace a node for a B-Tree. In general, these parameters can be optimized using a simple grid-search. Furthermore, it is possible to restrict the search space significantly based on best practices. For example, we found that a threshold of 128 or 256 (typical page-sizes for B-Trees) works well. Furthermore, for CPUs we can rarely afford neural nets with more 1 or 2 fully-connected hidden layers and 8 to 128

neurons per layer. Finally, given the rather low capacity of the models, it is possible to train the higher level model stages with small samples of the data, which significantly speeds up the training process.

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 impossible to learn data distribution, all models would be automatically replaced by B-Trees, making it virtually an entire B-Tree (there is some additional overhead between the stages, etc., but the performance is overall similar).

3.4 Search Strategies

To find the actual record in a leaf-page either binary search or scanning for small page-sizes with small payloads are usually the fastest strategies; despite many efforts, it was repeatedly reported that other search strategies do not provide much, if any, benefit because of their additional complexity [8]. Yet again, learned indexes might have an advantage here as well: the models actually predict the position of the key, which is likely to be much closer to the actual position of the record, while the min- and max-error is presumably larger. That is if we could leverage the fact that we have a good estimate of the position as part of the search within the min- and max-error around the position estimate, we might be able to find the record (or the lower key to the lookup key) faster than traditional binary search. We therefore developed several search strategies.

Model Binary 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 Search: This search strategy modifies our model binary search by not evenly splitting the range from the *middle* to the *left* and *right* position within every iteration. Rather the new middle is set depended on the standard deviation σ of the last stage model. For example, if the key is determined to be greater than the *middle*, the new *middle* is set to $min(middle + \sigma, (middle + right)/2)$.

Biased Quaternary Search: Finally we developed a new search strategy which in every iteration does not pick one new middle to test as in binary search, but three new data points, a so called quaternary search. The main reason why researchers tried quaternary search in the past is because of it presumable better pre-fetching behavior. That is, first all three "middle" points are calculated and requested to be prefetched by the CPU using intrinsics. Only afterwards the different "middle" points are tested and based on the result, the next iteration of the search is started very similar to binary search. This strategy is presumably better, if the CPU is able to fetch several data addresses in parallel from main memory, however, it was reported that in practice this strategy is mostly on par with binary search [8]. Yet, again having a better position estimate might help again: That is, we define our 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.

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 of length $\mathbf{x} \in \mathbb{R}^n$ where $\mathbf{x}_i$ is the ASCII decimal value (or Unicode decimal value depending on the strings). Further, most ML models 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. (There can be additional complexity in deeper networks that is independent of N.)

There are some interesting implications of this approach that demonstrate the difficulty in designing general ML models for CDFs of strings. If we consider the eight strings of length three that are the binary encoding of the integers $[0, 8)$ then we can easily model the position by $4\mathbf{x}_0 + 2\mathbf{x}_1 + \mathbf{x}_2$[4]. However, if we consider an encoding of the Unix dictionary, we see that the data is much more complicated. Almost three times as many words start with "s" as "e" making even how to model just the first character non-linear. Further, there are interactions between the characters – approximately 10% of words that start with "s" start with "sh" while only 0.1% of words starting with "e" start with "eh." DNNs, if wide or deep enough, can successfully model these interactions, and more commonly, recurrent neural networks (RNNs) have shown to be very successful in modeling text.

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 machine translation [59]. Additionally, there is significant research on feature selection to select the most useful subset of features thus limiting the number of features needed to be processed by the model. Further, GPUs and TPUs expect relatively large models, and as a result, will scale seamlessly in the string length as well as many of the more complex model architectures (e.g., recurrent and convolutional neural networks).

3.6 Results

In order to compare learned indexes with B-Trees, we created 4 secondary indexes over 3 real-world datasets, (1) Weblogs, (2) Maps [46], and (3) web-documents, and 1 synthetic dataset (4) Lognormal. The Weblogs dataset contains 200M log entries for every request to a major university web-site over several years and did an index over all unique timestamp. This data set 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 less irregularities than the weblog dataset. The web-document dataset consists of the 10M non-continuous document-ids of a large web index used as part of a real product at a large internet company. 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 1B. This data is of course highly non-linear, making the CDF more difficult to learn using neural nets.

For all datasets, we compare a B-Tree with different page sizes with learned indexes using a 2-stage RMI model and different second-stage sizes (i.e., 10k, 50k, 100k, and 200k) . Our B-Tree implementation is similar to the stx::btree but with further cache-line optimization and very competitive performance In a micro benchmark against FAST [36], a state-of-the-art SIMD optimized B-Tree we did not observe large differences. We tuned the 2-stage models using simple grid-search over rather simple models. That is, we only tried out 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

[4]In this example we assume the tokenization is normalized such that $\mathbf{x_i} = 0$ if character at position i is "0" and $\mathbf{x_i} = 1$ if character at position i is "1".

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

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

for the first stage work the best. For the second stage, it turned our that simple (0 hidden layers), which are essentially 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. Finally, all our learned index models were compiled with *LIF* and we only show numbers for the best performing models on an Intel-E5 CPU with 32GB RAM *without* GPU/TPUs over 30M lookups with 4 repetitions.

Load time: While the focus of this paper is not on loading or insertion time, it should be noted that most models can be trained rather quickly. For example, a model without hidden layers can be trained on over 200M records in just few seconds if implemented in C++. However, for more complex models, we opted to use Tensorflow, which, because of its overhead, took significantly longer. Yet, we are confident that we can grid-search over the hyperparameter space relatively quickly, likely on the order of minutes for simple models. Furthermore, auto-tuning techniques such as [52] could be used to further reduce the time it takes to find the best index configuration.

3.6.1 Integer Datasets

The results for the two real-world integer datasets (Maps and weblogs) and synthetic data (Lognormal) are shown in Figure 4, 5, and 6, respectively. As the main metrics we show the total lookup-time broken down into model execution (either B-Tree traversal or ML model) and local search time (e.g., to find the key in the B-Tree leaf-page). In addition, we report the index structure size (excluding the size of the sorted array), the space savings, and the model error with its error variance. The model error is the averaged standard error over all models on the last stage, whereas the error variance indicates how much this standard error varies between the models. Note, for B-Trees it is always a fixed error dependent on the page-size as there are no stages. The color-encoding in the speedup and size columns indicates how much faster or slower (larger or smaller) the index is against the baseline of the cache-optimized B-Tree index with a page-size of 128.

As can be seen, the learned index dominates the B-Tree index in almost all configurations by being up to 3× faster and being up to an order-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 not only orthogonal but for neural nets even more compression potential exist. 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

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

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

referred to as quantization). In contrast to B-Tree compression techniques this could actually speed up the computation even further as well.

Interesting is also that quarternary search only helps for some datasets. For example, it does help a little bit in the weblog and log-normal datasets, but not in the maps dataset. We did not report on the biased search or show the results of different search strategies for B-Trees as they didn't provide any benefit for the numeric datasets. It is also interesting to note, that the model accuracy also varies widely. Most noticeable for the synthetic dataset and the weblog data the error is much higher, which influences the search time. As a comparison point, we also show the learned index with a more complex 1st stage model ("Learned Index Complex"), which is across the board able to significantly reduce the error with 2 fully-connected hidden layers and 16 neurons per layer. Yet, on the CPU the model complexity does not pay off (i.e., the model execution time is too high to justify the shorter search time). However, as argued earlier with GPU/TPUs this trade-off is going to change and we speculate much more complex models will be beneficial on the next generation of hardware.

It can be observed that the second stage size has a significant impact on the index size and lookup performance. This is not surprising as the second stage determines how many models have to be stored. Worth noting is that our second stage uses 10,000 or more models. This is particularly impressive with respect to the analysis in Section 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 did not report on any hybrid models for the integer data set as they, like biased search, did not provide any benefit.

3.6.2 String Data Sets

The results for the string-based document-id dataset are shown in Figure 7. This time we included hybrid indexes and show the different search strategies in a separate Figure 8. However, we did include our best model in the table which is a non-hybrid RMI model index with quaternary search, named "Learned 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.

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

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

As can be seen, the speedups for learned indexes over B-Trees for strings is not as prominent anymore. This has partially to do with the fact, that the model execution is rather expensive, a problem GPU/TPUs would solve. Furthermore, searching over strings is much more expensive thus higher precision often pays off. That is the reason that hybrid indexes, which replace bad performing models through B-Trees actually help to improve performance.

Finally, because of the cost of searching, the different search strategies make a bigger difference as shown in Figure 8 for a 2 stage neural net model with one or two hidden layers. Note, that we do not show different search strategies for B-Trees as they did not improve performance. The reason why biased search and quaternary search performs better is that they can take the standard error into account.

3.7 Future Research Challenges

So far our results focused on index-structures for read-only in-memory database systems. As we already pointed out, the current design, even without any significant modifications, is already useful to replace index structures as used in data warehouses, which might be only updated once a day, or BigTable [18] where B-Trees are created in bulk as part of the SStable merge process. In this section, we outline how the idea of learned index structures could be extended to even insert-heavy workloads.

3.7.1 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 on order table. For the moment we focus on the latter and consider the approach of introducing additional space in our sorted dataset, similar to a B-Tree which introduces space in its representation through its min- and max-fill factor for every page. However, in contrast to B-Trees, assume that we would not spread the space evenly but make it dependent on the learned cumulative density function. Finally, assume that the inserts follow roughly a similar pattern as the learned CDF; not an unreasonable assumption as in our example of the secondary index over customer-ids, it would

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

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

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

Figure 8: Learned String Index with different Search Strategies

only mean that customers roughly keep their shopping behavior. Under these assumptions the model might not need to be retrained at all. Instead the index "generalizes" over the new items and inserts become an $O(1)$ operation as they can be put directly into the free space, and space is available where it is needed the most. In contrast, B-Trees require $O(\log n)$ operations for inserts for finding and re-balancing the tree (especially if inserts in a certain region are more common than in others). Similar, for *append inserts* a model might also not require relearning if the model is able to learn the key-trend for the new items.

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 $O(logn)$ lookup 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. Consider a simple linear model: in that case an inserted item can never increase the error by more than $max\_abs\_error + 1$. Furthermore, retraining a linear model has the cost of $O(K_{M+1})$ where K_{M+1} is the number of items the model covers in the data. If the linear model is not sufficient to achieve an acceptable error, we can split the range in two models, and retrain the model in the stage above. This model again might require $O(K_M)$ to be retrained, where K_M is the number of models on the last stage M and so on, after we retrained the first model. Thus, up to the retraining of the first model, it is easy to bound the absolute error. However, in order to provide still a good error for the very first model, we might be required to increase the capacity of the model, for example, by adding a polynomial or an additional hidden layer. Here it might be possible to bound again the required increase in complexity through techniques like VC-dimensions. Studying these implications especially for other types of models, including NN, is left for future work.

An alternative much simpler approach to handling inserts is to build a delta-index [49]. 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 [18] and many other systems, and also has the advantage that for large

retraining operations specialized hardware, such as GPU/TPUs, could be used, which would significantly speed-up the process even if an entire model retraining is required.

Finally, it is possible to warm-start every model training by using the previous solution as a starting point. Especially models which rely on gradient-descent optimization can profit from this optimization [33].

3.7.2 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 $p = F(X < \text{Key}) * N$ is violated. In the following we outline several options to overcome this issue:

Leveraging the RMI structure: The RMI structure already partition the space into regions. With small modifications to the learning process it, 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. This way we can simply store an offset to the models, which refers to the position the data is stored on disk.

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 only work if the disk pages are very large, maybe in the hundreds of megabytes if not gigabytes as otherwise the translation table becomes too 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.

4 Point Index

Next to range indexes, Hash-maps for point lookups play a similar or even more important role in DBMS. Conceptually Hash-maps use a hash-function to deterministically map keys to random positions inside an array (see Figure 9(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. The number of expected conflicts can be derived similar to the birthday paradox and in expectation would be around 33% or $33M$ slots. For each of these conflicts, the index has to either use a linked-list to handle the "overflow" (see Figure 9(a)) or use some form of secondary probing. Both can have significant overhead – for larger data sets traversing a linked-list or secondary probing yields most likely to a cache-miss, costing up to 50-100 cycles. Many variants and combinations of these two approaches exist (e.g., [1, 20]), which all have the goal of mitigating the effect of conflicts. Therefore, most solution often allocate significant more memory (i.e., slots) than records to store as well as combine it with additional data structures (such as dividing the hash-space into buckets), which yet again requires more overhead. For example, it is reported that Google's Dense-hashmap has a typical overhead of of about 78% memory (i.e., if your data takes up X bytes, the Hash-map uses .78X more bytes in overhead) , whereas Google's sparse-hashmap only has 4 bits overhead but is up to 3-7 times slower because of its search and data placement strategy [2].

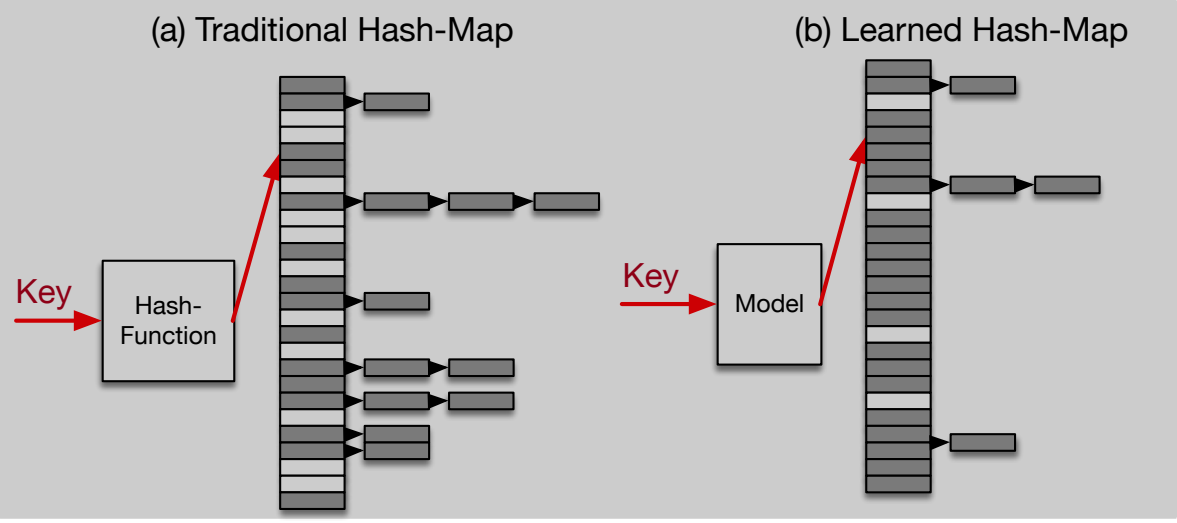


Figure 9: Traditional Hash-map vs Learned Hash-map

Similar to the B-Trees, machine learned models might provide a solution. For example, if we could learn a model which uniquely maps every key into a unique position inside the array we could avoid conflicts (see Figure 9(b)). While the idea of learning models as a hash-function is not new, existing work mainly focused on learning a better hash-function to map objects from a higher dimensional space into a lower dimensional space for similarity search (i.e., similar objects have similar hash-values) [55, 57, 32]. To our knowledge it has not been explored if it is possible to learn models which yield more efficient point index structure. While on first sight it seems impossible to speedup Hash-maps using machine learning models, if memory space is a significant concern it might actually be an option. As our experiments show, a high utilization (less than 20% wasted space at small overhead per record) for traditional techniques is extremely hard to achieve and results in significant performance penalties. In contrast learned models are capable of reaching higher utilization depending on the data distribution. Furthermore, it is yet again possible to offload the models to GPU/TPUs, which might mitigate the extra cost of executing models over hash-functions.

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 CDF, no conflicts would exist. Furthermore, the hash-function is orthogonal to the actual Hash-map implementation and can be combined with the linked-list or any other approach.

For the model architecture, 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 architecture and dataset.

Note, that inserts for hash-model indexes are done in the same way as for normal Hash-maps: the key is hashed with $h(k)$ and the item is inserted in the returned position. If the position is already taken, the Hash-map implementation determines how to handle the conflict. That implies, as long as the inserts follow a similar distribution as the existing data, the learned hash-function keeps its efficiency. However, if the distribution changes, the model might have to be retrained as outlined before. Again, this goes beyond the scope of this paper.

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

Figure 10: Model vs Random Hash-map

4.2 Results

To test the feasibility of Hash-Indexes using machine learning models, we implemented a linked-list based Hash-map; records are stored directly in the Hash-map and only in the case of a conflict the record is 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 lookup performance. 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 inplace 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 linked-list 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 randomized hash-function that only uses two multiplications, 3 bitshifts and 3 XORs, which is much faster than, for example, cryptographic hash functions. As our model hash-functions we used the same 2-stage RMI models with 100k models on the second stage as in the previous section, without any hidden layers. We did not try any other models as we are particular interested in fast lookup speed.

We used the same three int datasets as from the previous section and 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 size, minimizes the empty slots within the Hash-map at the expense of longer linked lists.

The results are shown in Figure 10 listing the average lookup time for a key, the number of empty slots in GB as well as percent of the total number of available slots and the space improvement. Note, that in contrast to the previous section, we *do include the data size*. The main reason is, that in order to enable 1 cache-miss lookups, 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 the Figure, the index with the model hash function overall has similar performance

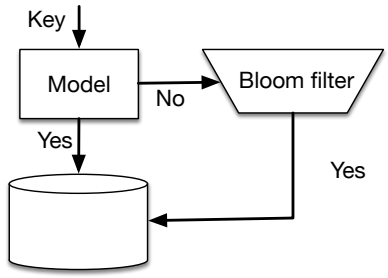


Figure 11: Bloom filters as a classification problem

while utilizing the memory better. For example, with 100% slots (i.e., the number of slots in the Hash-map matches the data size), randomized hashing always experiences around 35% conflicts (the theoretical value is 33.3%) and waste $1.5GB$ of main memory, whereas the learned hash functions better spread out the key-space and thus are able to reduce the unused memory space by up to 80%, depending on the dataset.

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 has still an advantage because of the still prevalent birthday paradox. We also did experiment with even smaller sizes and observed that around 50% size there is no noticeable difference anymore between the random and learned hash function. However, reducing the Hash-map size to only contain half of the slots of the data size also significantly increases the lookup time as the linked lists grow.

4.3 Alternative Approaches and Future Work

Next to using CDFs as better hash-functions, it might also be possible to develop other types of models. While outside the scope of this paper, we explored to some extent the idea of co-optimizing the data placement and the function to find the data. However, in many cases it turned out that yet again experts with a mixture of models for many data sets performed the best.

Furthermore, it might be possible to more tightly integrate the Hash-map with the model as done with the Hybrid indexes. That is, we could learn several stages of models and replace the poor performing models (i.e., the ones which create more conflicts than randomized hashing) with simple randomized hash-functions. This way, the worst case performance would be similar to randomized hashing. However, if the data has patterns which can be learned, a higher memory utilization could be achieved.

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 exist on cold storage. For example, BigTable uses them to determine if a key is contained in an SSTable [18].

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 Figure12(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 potential *false positives*.

While Bloom filters are highly space-efficient, they can still occupy a significant amount of memory. For

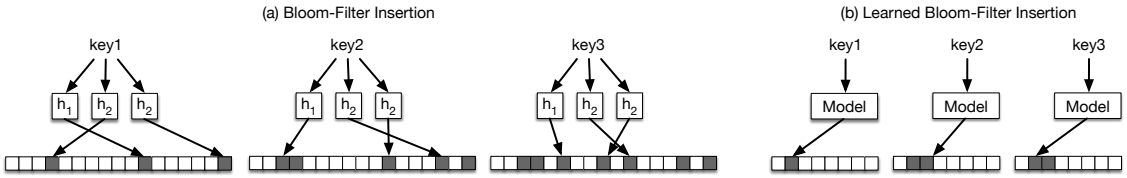


Figure 12: Bloom-filters with learned hash-functions

example for 100M records with a targeted false positive rate (FPR) of 0.1% requires roughly $14\times$ more bits than records. So 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 [43], 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.

In the following we outline two potential ways to build existence indexes using learned models.

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 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.

Traditionally, existence indexes make no assumption or use of the distribution of keys nor how they differ from non-keys. 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 – this could be either randomly generated keys, based on logs of previous queries to the existence index, or generated by a machine learning model [26]. We denote the set of keys by $\mathcal{K}$ and the set of non-keys by $\mathcal{U}$. While Bloom filters guarantee a specific false positive rate (FPR) and a false negative rate (FNR) of zero for any set of queries, we follow the notion that we want to provide a specific FPR for realistic queries in particular and maintain a FNR of 0.

5.1.1 Bloom filters as a Classification Problem

Another way to frame the existence index is as a binary classification task. That is, we want to learn a model f that can predict if a query x is a key or non-key. To do this, we train a neural network 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,

$$L = \sum_{(x,y)\in\mathcal{D}} y \log f(x) + (1-y)\log(1-f(x)). \tag{2}$$

As before, f can be chosen to match the type of data being indexed. We generally consider recurrent neural networks (RNNs) or convolutional neural networks (CNNs), as they have been repeatedly shown to be effective in modeling strings [54, 29].

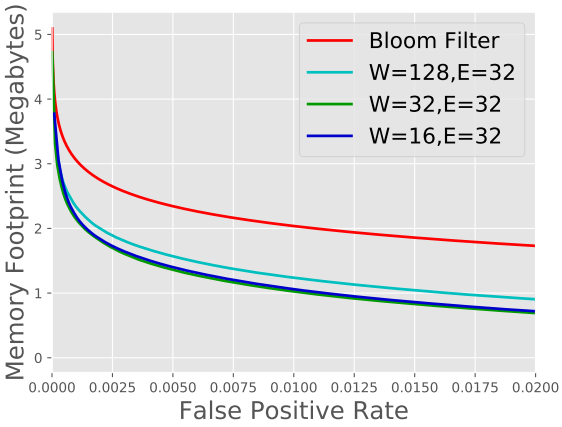


Figure 13: Learned Bloom filter improves memory footprint at a wide range of false positive rates. (Here W is the RNN width and E is the embedding size for each character.)

Given a trained model f, how do we make it an effective existence index? The output of $f(x)$ can be interpreted as the probability that x is a key in our database; we must choose a threshold τ above which we will assume that the key exists in our database. As Bloom filters are generally tuned for a particular FPR, we can set τ to achieve the desired FPR on a held-out dataset of queries to the index. 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 11: if $f(x) \geq \tau$, the key is believed to exist; otherwise, check the overflow Bloom filter.

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 linearly with the size of key set, the overflow Bloom filter will scale with the FNR. That is, even if we have a false negative rate of 50%, we will *halve* the size of the Bloom filter (excluding the model size). 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 like GPUs and TPUs, 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

In the classification setup, we choose a threshold τ and accept that predictions with $f(x) \geq \tau$ will have a non-zero FPR and predictions with $f(x) < \tau$ will have a non-zero FNR. This contradicts the typical perspective on hash functions in Bloom filters in that no slot should have a non-zero FNR. We can use f as a hash function of this sort by using it to map to a bit array of size m. Because f, as learned above, maps queries to the range $[0, 1]$, we can assume some discretization of that space by a function d. For example, we can most simply define d by $d(p) = \lfloor mp \rfloor$. As such, we can use $d(f(x))$ 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.

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 train a character-level RNN (GRU [19], in particular) to predict which set a URL belongs to.

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. We find that if we want to enforce a 1% FPR, the TNR is 49%. As described above, the size of our Bloom filter scales with the FNR (51%). As a result, we find that our model + the spillover Bloom filter uses 1.07MB, a 47% reduction in size. If we want to enforce an FPR of 0.1%, we have a FNR of 71%, which brings the total Bloom filter size down from 3.06MB to 2.2MB, a 28% reduction in memory. We observe this general relationship in Figure 13. Interestingly, we see how different size models balance the accuracy vs. memory trade-off differently.

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, 13]. 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.

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 [28], such as B+-Trees [15] for disk based systems and T-trees [38] or balanced/red-black trees [14, 17] 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 [47]. Similarly, there has been work on making use of SIMD instructions such as FAST [36] or even taking advantage of GPUs [36, 50, 35]. Moreover, many of these (in-memory) indexes are able to reduce their storage-needs by using offsets rather than pointers between nodes. There exist also a vast array of research on index structures for text, such as tries/radix-trees [16, 37, 23], or other exotic index structures, which combine ideas from B-Trees and tries [39].

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 [28, 25, 45], or hybrid hot/cold indexes [61]. 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 [24], BF-Trees [12], and B-Tree interpolation search [27]. BF-Tree uses a B+ tree to store information about a region of the dataset, instead of the indexing individual keys. However, leaf nodes in a BF-Tree 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 [27] 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 [60], Block Range Indexes [53], and Small Materialized Aggregates (SMAs) [44] all store information about value ranges but again do not take advantage of the underlying

properties of the data distribution.

Better Hash-Functions: Similar to the work on tree-based index structures, there has been a lot of work on hash-maps and hash-functions [40, 57, 56, 48]. Most notably, there has been even work on using neural networks as a hash-function [55, 57, 32]. However, this work is quite distinct from learning models to build more efficient hash-maps as it mainly focuses on mapping a large dimensional space to a smaller one for similarity search [57] or to create better features for machine learning, a.k.a. feature hashing [58]. Likely the closest work related to our idea of building a better hash function is [55], which tries to use neural nets to learn a stronger cryptographic hash function. However, its goal is still different from our goal of mapping keys distinctly to a restricted set of slots within a hash-map.

Bloom-Filters: Finally, our existence indexes directly builds upon the existing work in bloom-filters [22, 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 [31, 30]. 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 as 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 cumulative distribution function (CDF). Estimating the CDF is non-trivial and has been studied in the machine learning community [41] with a few applications such as ranking [34]. However, most research focuses on modeling the probability distribution function (PDF) leaving many open questions for how to effectively model the CDF.

Mixture of Experts: Our RM-I architecture follows a long line of research on building experts for subset of the data [42]. With the growth of neural networks, this has become more common and demonstrated increased usefulness [51]. 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 showed that learned indexes can provide significant benefits by utilizing the distribution of data being indexed. This opens the door to many interesting research questions.

Multi-Dimensional Indexes: Arguably the most exciting research direction for the idea of learned indexes are to extend them to multi-dimensional index structures. Models, especially neural nets, 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 viable. 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 database indexes, and we believe this is a fruitful direction for future research.

References

[1] Google's sparsehash. `https://github.com/sparsehash/sparsehash-c11`.

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

[3] An in-depth look at googles 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] NVIDIA TESLA V100. `https://www.nvidia.com/en-us/data-center/tesla-v100/`.

[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 *Proceedings of the anti-phishing working groups 2nd annual eCrime researchers summit*, pages 60–69. ACM, 2007.

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

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

[13] 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.

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

[15] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, pages 107–141, New York, NY, USA, 1970. ACM.

[16] M. Böhm, B. Schlegel, P. B. Volk, U. Fischer, D. Habich, and W. Lehner. Efficient in-memory indexing with generalized prefix trees. In *Datenbanksysteme für Business, Technologie und Web (BTW), 14. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 2.-4.3.2011 in Kaiserslautern, Germany*, pages 227–246, 2011.

[17] 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.

[18] 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 *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 205–218, 2006.

[19] 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 *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1724–1734, 2014.

[20] J. G. Cleary. Compact hash tables using bidirectional linear probing. *IEEE Trans. Computers*, 33(9):828–834, 1984.

[21] 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.

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

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

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

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

[26] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

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

[28] G. Graefe and P. A. Larson. B-tree indexes and CPU caches. In *Proceedings 17th International Conference on Data Engineering*, pages 349–358, 2001.

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

[30] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 841–850, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

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

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

[33] G. E. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *CoRR*, abs/1503.02531, 2015.

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

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

[36] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 339–350, New York, NY, USA, 2010. ACM.

[37] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner. Kiss-tree: Smart latch-free in-memory indexing on modern architectures. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN '12, pages 16–23, New York, NY, USA, 2012. ACM.

[38] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, pages 294–303, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[39] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 38–49, Washington, DC, USA, 2013. IEEE Computer Society.

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

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

[42] D. J. Miller and H. S. Uyar. A mixture of experts classifier with learning based on both labelled and unlabelled data. In *Advances in Neural Information Processing Systems 9, NIPS, Denver, CO, USA, December 2-5, 1996*, pages 571–577, 1996.

[43] M. Mitzenmacher. Compressed bloom filters. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing, PODC 2001, Newport, Rhode Island, USA, August 26-29, 2001*, pages 144–150, 2001.

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

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

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

[47] J. Rao and K. A. Ross. Making b+- trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 475–486, New York, NY, USA, 2000. ACM.

[48] S. Richter, V. Alvarez, and J. Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. *Proc. VLDB Endow.*, 9(3):96–107, Nov. 2015.

[49] D. G. Severance and G. M. Lohman. Differential files: Their application to the maintenance of large data bases. In *Proceedings of the 1976 ACM SIGMOD International Conference on Management of Data*, SIGMOD '76, pages 43–43, New York, NY, USA, 1976. ACM.

[50] A. Shahvarani and H.-A. Jacobsen. A hybrid b+-tree as solution for in-memory indexing on cpu-gpu heterogeneous computing platforms. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1523–1538, New York, NY, USA, 2016. ACM.

[51] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[52] E. R. Sparks, A. Talwalkar, D. Haas, M. J. Franklin, M. I. Jordan, and T. Kraska. Automating model search for large scale machine learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015*, pages 368–380, 2015.

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

[54] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[55] M. Turcanik and M. Javurek. Hash function generation by neural network. In *2016 New Trends in Signal Processing (NTSP)*, pages 1–5, Oct 2016.

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

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

[58] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 1113–1120, New York, NY, USA, 2009. ACM.

[59] 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.

[60] J. Yu and M. Sarwat. Two Birds, One Stone: A Fast, Yet Lightweight, Indexing Scheme for Modern Database Systems. In *VLDB*, pages 385–396, 2016.

[61] 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 *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1567–1581, 2016.