Learned Index Structures

Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, Neoklis Polyzotis

ABSTRACT

Indexes are models: a BTree-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 talk, 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. Our initial results show, that by using simple 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 might have significant implications for future systems designs.

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, for range requests (e.g., retrieve all records in a certain timeframe) B-Trees are the best choice. If data is only looked up by a key, Hash-Maps are hard to beat in performance. In order to determine if a record exists, an existence index like a Bloom-filter is typically used. Because of the importance of indexes for database systems and many other applications, they have been extensively tuned over the last decades to be more memory, cache and/or CPU efficient [3, 6, 2, 1].

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 integers keys (e.g., the keys 1 to 100M), one would not need to use a conventional B-Tree index over the keys since they key itself can be used as an offset, making it a constant $O(1)$ 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 $O(1)$. Of course, in most real-world use cases the data does not perfectly follow a known pattern, and it is usually not worthwhile to engineer a specialized index for every use case. However, if we could learn a model, which reflects the data patterns, correlations, etc. of the data, it might be possible to automatically synthesize an index structure, a.k.a. a **learned index**, that leverages these patterns for significant performance gains.

In this talk, we will explain to what extent learned models, including neural networks, can be used to replace traditional index structures from Bloom-Filters to B-Trees. 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. We argue that none of these obstacles are as problematic as they might seem with potential huge benefits, such as opening the door to leveraging hardware trends in GPUs and TPUs.

In the following we only outline, why B-Trees can be replaced with learned models. We refer to our technical report at [4] for a more in depth discussion why other index structures can also be replaced using models.

2. RANGE INDEX

In the case of range indexes, the data is stored in sorted order and an index is built to find the starting position of the range in the sorted array. Once the position at the start of the range is found, the database can merely walk through the ordered data until the end of the range is found. (We do not consider inserts or updates for now.)

2.1 Background

The most common index structure for finding the position of a key in the sorted array is a B-Tree. B-Trees are balanced and cache-efficient trees. They differ from more common trees, like binary trees, in that at each node has a fairly large branching factor (e.g., 100) to match the page size for efficient memory hierarchy usage (i.e., the top-k nodes of the tree always remain in cache). As such, for each node that is processed, the model gets a precision gain of 100. Of course, processsing a node takes time. Typically, traversing a single node of size 100, assuming it is in cache, takes approximately 50 cycles to scan the page (we assume scanning, as it is usually faster than binary search at this size).

2.2 Learned Ranged Index

At the core, B-Trees are models of the form $f(\text{key}) \rightarrow \text{pos}$. We will use the more common machine learning notation where the key is represented by x and the position is y. Because we know all of the data in the database at training time (index construction time), our goal is to learn a model $f(x) \approx y$. Interestingly, because the data x is sorted, f is modeling the cumulative distribution function (CDF) of data, a problem which has received some attention [5][1].

2.2.0.1 Model Architecture.

As a regression problem, we train our model $f(x)$ with squared error to try to predict position y. The question is what model architecture to use for f. As outline above, for the model to be successful, it needs to improve the precision of predicting y by a factor greater than 100 in a less than 50 CPU cycles. With often many million of examples and needing high accuracy, building one large wide and deep

[1]This assumes records are fixed-length. If not, the function is not the CDF, but another monotonic function.

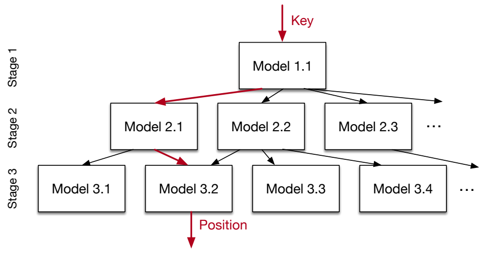


Figure 1: Staged models

neural network often gives not very strong accuracy while being expensive to execute. Rather, inspired by the mixture of experts work [7], we build a hierarchy of models (see Figure 1). We consider that we have $y \in [0, N)$ and at stage ℓ there are M_ℓ models. We train the model at stage 0, $f_0(x) = y$. As such, model k in stage ℓ, denoted by $f_\ell^{(k)}$, is trained with loss:

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

Note, we use here the notation here of $f_{\ell-1}(x)$ recursively executing $f_{\ell-1}(x) = f_{\ell-1}^{\left(f_{\ell-2}(x)\right)}(x)$. Therefore, in total, we iteratively train each stage with loss L_ℓ to build the complete model. Interestingly, we find that it is necessary to train iteratively as Tensorflow has difficulty scaling to computation graphs with tens of thousands of nodes.

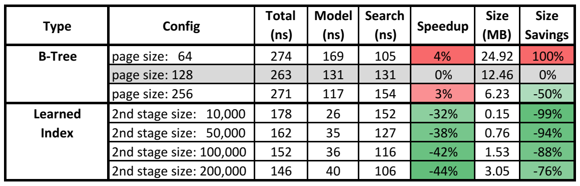
2.2.0.2 *Constraints.*

Unlike typical ML models, it is not good enough to return the approximate solution. Rather, at inference time, when the model produces an estimate of the position, we must find the actual record that corresponds to the query key. Traditionally, B-Trees provide the location of the beginning of the page where the key lies. For learned indexes, we need to search around the prediction, to actually find the beginning of the range.

Interestingly, we can easily bound the error of our model such that we can use classic search algorithms like binary search. This is possible because we know at training time all keys the model is indexing. As such, we can compute the maximum error Δ the model produces over all keys, and at inference time perform a search in the range $[\hat{y} - \Delta, \hat{y} + \Delta]$. Further, we can consider the model's error over subsets of the data. For each model k in the last stage of our overall model, we compute its error Δ_k. This turns out to be quite helpful as some parts of the model are more accurate than others, and as the Δ decreases, the lookup time decreases. An interesting future direction is to minimize worst-case error Δ rather than average error. In some cases, we have also built smaller B-Trees to go from the end of the model predictions to the final position. These hybrid models are often less memory efficient but can improve speed due to cache-efficiency.

2.3 Experiments

We have run experiments on 200M web-log records with complex patterns created by school holidays, hourly-variations, special events etc. We assume an index over the timestamps of the log records (i.e., an index over sorted, unique 32bit timestamps). We used a for read-workloads cache-optimized B-Tree with a page-size of 128 as the baseline, but also evaluated other page sizes. The B-Tree is dense and does not have any free space for inserts. As the learned range index we used a 2-stage model (1st stage consist of 1 linear model, 2nd stage consist of varying number of linear models). We evaluated the indexes on an Intel-E5 CPU with 32GB RAM *without* a GPU/TPU. Results of our experiments can be seen in Figure 2. We find that learned indexes are significant faster than B-Tree models while using much less memory. This results is particular encouraging as the next generation of TPUs/GPUs will allow to run much larger models in less time (though the invokation time of TPUs/GPUs might be a challenge).



Type	Config	Total (ns)	Model (ns)	Search (ns)	Speedup	Size (MB)	Size Savings
B-Tree	page size: 64	274	169	105	4%	24.92	100%
	page size: 128	263	131	131	0%	12.46	0%
	page size: 256	271	117	154	3%	6.23	-50%
Learned Index	2nd stage size: 10,000	178	26	152	-32%	0.15	-99%
	2nd stage size: 50,000	162	35	127	-38%	0.76	-94%
	2nd stage size: 100,000	152	36	116	-42%	1.53	-88%
	2nd stage size: 200,000	146	40	106	-44%	3.05	-76%

Figure 2: Performance learned index vs B-tree.

3. FUTURE DIRECTIONS AND CONCLUSION

In our arxiv paper [4] we describe how other index structures, such as BloomFilters or HashMaps, can also be enhanced using machine learning models. We believe this perspective opens the door for numerous new research directions ranging from new indexes which can better leverage GPUs to novel index structures for multi-dimensional data up to extending the idea to other algorithms/data structures, such as sorting.

Acknowledgements: We would like to thank Chris Olston for his feedback during the development of this work.

4. REFERENCES

[1] 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.
[2] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *CoNEXT*, 2014.
[3] G. Graefe and P. A. Larson. B-tree indexes and cpu caches. In *ICDE*, pages 349–358, 2001.
[4] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The Case for Learned Index Structures. *ArXiv e-prints*, Dec. 2017.
[5] M. Magdon-Ismail and A. F. Atiya. Neural networks for density estimation. In M. J. Kearns, S. A. Solla, and D. A. Cohn, editors, *NIPS*, pages 522–528. 1999.
[6] 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.
[7] 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.