Short Paper

Interpolation-friendly B-trees: Bridging the Gap Between Algorithmic and Learned Indexes

Ali Hadian
Imperial College London
hadian@imperial.ac.uk

Thomas Heinis
Imperial College London
t.heinis@imperial.ac.uk

ABSTRACT

Index structures such as B-trees and bloom filters are the "petrol engines" of database systems, but these structures do not fully exploit patterns in data distribution. To address this, researchers have suggested using machine learning models as "electric engines" that can entirely replace index structures. Such a paradigm shift in data system design, however, opens many unsolved design challenges, e.g. more research is needed to understand the theoretical guarantees and design efficient support for insertion and deletion.

In this paper we adopt a different position: index algorithms are good enough, and that instead of going back to the drawing board to fit data systems with learned models, we should develop lightweight "hybrid engines", where a *helping model* "boosts" classical index performance using techniques from machine learning.

As a case study, we show how interpolation techniques can be integrated with a B-trees with negligible change to the structure and memory footprint of the base algorithm. We show that such a simple helping model, called Interpolation-Friendly B-tree (IFB-tree), can boost the speed of B-trees by up to 50%.

1 INTRODUCTION

Data engines exploit efficient implementations of algorithmic index structures, such as hash tables, B-trees, radix-trees, bloom filters, etc. Index structures fit different tasks and workloads, e.g. B-trees are efficient for range lookups and hash tables are the tool of choice for point queries. Different aspects for each of these indexes have been studied for decades. Many tuning suggestions exist, for example, for B-trees to efficiently fit hardware specifications including the latency/bandwidth ratio and cache-sizes, and various extensions have been suggested to improve the performance even further. [4, 5, 11, 12].

Recently, it was suggested that general-purpose index structures such as B-tree cannot exploit common patterns in data distribution of the real-world data, hence proposing the use of machine learning (ML) models [9]. In this approach, a learned model entirely replaces a classical index and learns how to perform the same behavior. For example, a B-tree can be replaced by a learned index (based on deep learning models) that takes the key as input and *estimates* the position of the corresponding data record in a sorted set, i.e., a clustered index or sorted list of keys.

Learned indexes can be effective for read-only lookups over many data distributions. However, lack of theoretical performance guarantees for a learned model and the challenges for handling update operations in a learned model has lead to an extensive debate in the community. In general, and similar to the analogy of 'petrol vs electric' engines, adopting machine learning techniques could yield elegant methods in data management but the design and maintenance of an ML-enhanced DBMS opens numerous challenges that require comprehensive research. In the meantime, we need hybrid engines that do not render current algorithms and indexes unnecessary. We refer to the hybrid engines as "helping models". Figure 1 illustrates a classical B-tree index (a) vs a learned index (b), the hybrid approach (c) where helping models improve a classical index on different stages.

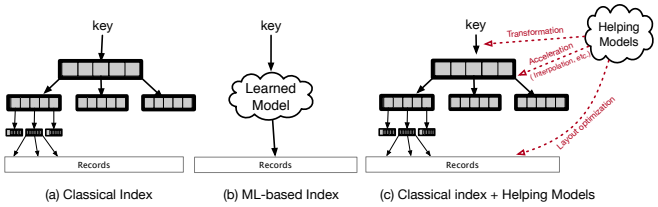


Figure 1: Alternatives for classical indexes

In this work, we try to bridge the gap between traditional index structures and the ML approach, suggesting that a helping model can be *integrated* with a traditional index without incurring undue overhead. We suggest that the ability of an ML model to make *distribution-aware* indices is not a rival for classical indexes, but indeed complements them.

We therefore present the *Interpolation-Friendly B-tree* (IFB-tree), which sticks with the traditional B-tree structures to enjoy their performance guarantees, yet is able to exploit the basic ideas of a learned index such as *bounding error-windows* for intra-node lookups. More specifically, we demonstrate how linear interpolation can be integrated with a B-tree index to reduce unnecessary operations with a negligible memory footprint. The process can be done by analyzing a B-tree and labeling *interpolation-friendly* nodes, so that further access to such nodes can be accelerated using interpolation. Our experiments show that the Interpolation-Friendly B-trees (IFB-trees) gives up to 50% improvement over B-trees.

2 TO B-TREE OR NOT TO B-TREE?

2.1 B-tree overview

The B-tree is a generic data structure. State-of-the-art B-trees are n-ary binary trees, i.e., generic data structures that do not assume any specific pattern in the underlying key distribution [4, 5, 11, 12]. B-tree lookup time consists of the time to search within each node, plus the time to follow the pointers and load the next nodes in bottom levels. This takes time and requires keeping a portion of data (inner nodes) in memory.

One of the key issues in a B-tree is that, to locate an item in each of the B-tree nodes, the entire node should be searched, either by linear scan or binary search.

In the following, we consider the alternative approaches that provide "smarter" methods for looking up keys in B-trees. The general solution for a more targeted lookup in a B-tree node (i.e., a sorted list of keys) is to use a method that effectively leverages the underlying key distribution.

2.2 Learned indexes

Machine learning provides various tools for learning a distribution. Recently, Kraska et al. [9] suggested that if a machine learning model can fully learn the CDF of the key distribution, it can directly predict the location of the queried keys in the table pages. They exploited a hierarchy of deep learned models to help shrink the problem space. Since all predictions in machine learning are susceptible to errors, the *maximum error* for the models needs to be computed in advance. Once the position is estimated, we can scan a *range* around the estimated position in the table pages. This is called the *error window*, i.e. $[\hat{pos}(key) - MaxErr_{\text{Left}}\ ,\ \hat{pos}(key) + MaxErr_{\text{Right}}]$. As shown in Figure 1(a,b), a learned index entirely replaces a classical data structure (like B-tree). If the prediction error is small enough, a learned model can ultimately beat B-tree's lookup time.

Challenges of learned indexes. A learned index can learn many complex distributions and locate the position of the key with a considerably low error margin. The idea of ML-indexes is be very effective for modeling some specific CDFs, yet ML models are not efficient for every key distribution. Moreover, state-of-the-art learned models are mostly suitable for read-only data. Despite that these models can handle a few insertions and deletions by reserving empty spaces in the sorted list, more update requests require re-training the model, which is computationally expensive. Since the model is bound to the mapping between the key's value and the position's offset, any changes to this mapping, such as inserts and deletes, make the learned model ineffective. Moreover, learned indexes require that the table pages be stored in a continuous block of memory so that records can be retrieved once the position is estimated. This assumption does not hold when the keys are not sorted, e.g. in a secondary (non-clustered) index, hence limits the applicability of a learned index.

2.3 IFB-tree

We believe that the main issue behind the challenges in a learned index is that it "replaces" an algorithmic data structure, but cannot deliver all the same operations. Alternatively, we suggest that a similar ideas of a learned index can be embedded within a data structure, such as the B-tree. Unlike the ML index approach which entirely replaces conventional index structures with a learned model over the sorted page of keys, we stick with the genuine structure of the B-tree. In fact, the B-tree itself is very effective in modeling the CDF of the key distribution by repeatedly breaking it into smaller parts. We consider the fact that a considerable share of B-tree lookup time is spent on intra-node search, i.e., to find the smallest value in a B-tree node that is larger (or equal) to the query point q. If we manage to outperform the intra-node search in a B-tree, the efficiency of B-trees can be improved without changing its theoretical performance guarantees.

Learning node distribution. Since keys are sorted in each B-tree node, we can think of a smarter lookup instead of naively doing a linear or binary search over all keys in the node. Similar to the learned indexes, a tiny model can predict the position of a key in each node, which reduces the search space within each B-tree node. However, even the simplest models (e.g. linear regression) requires keeping model parameters for each node in memory, and the cost of managing and loading the parameters does not lead to any performance improvements.

Node interpolation. Following the general idea of ML indexes, our solution is to estimate a range that guarantees the target key resides in. While *interpolation* looks like a naive approach, it could be very effective and has near-zero cost: it simply requires one bit per key, and the computation is very fast and simple. Figure 2b illustrates how interpolation can be done in a B-tree node. If the queried key v_q is between the two keys in a B-tree node, say v_i, v_{i+1} , it should be located on the next level, which could be the next node in the tree or a node in the physical table pages if the current node is a leaf. The interpolated index of the entry corresponding to v_q is: $\hat{p_q} = \left\lfloor \frac{v_q - v_i}{v_{i+1} - v_i} \times \texttt{node\_size} \right\rfloor$

The interpolation has an error, unless the keys follow an exact arithmetic progression, which is almost never the case in real-world except in case of auto-generated key sequences. Let's assume the maximum error for any value in a node is $\text{MaxErr}_{\text{Left}}$ and $\text{MaxErr}_{\text{Right}}$, respectively. This means that the area between $\hat{p_q} - MaxErr_{\text{Left}} + \hat{p_q} + MaxErr_{\text{Right}}$ should be scanned to find the actual position corresponding to v_q.

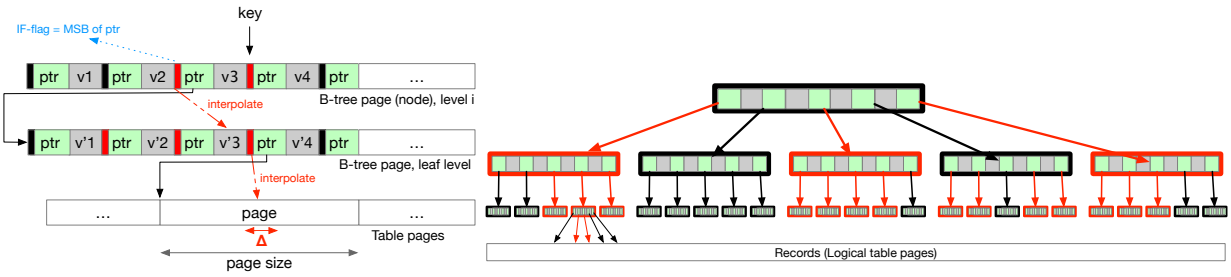
Global error window. Storing the errors values $\text{MaxErr}_{\text{Left}}$ and $\text{MaxErr}_{\text{Right}}$ alongside the keys in the B-tree node will increase the memory footprint and hence greatly reduces the performance. Moreover, having variable error values for each node makes it challenging to exploit the SIMD capabilities for interpolation and search. We prefer not to store any extra information in each node and keep the structure of B-tree untouched to the extent possible. Therefore, we define a *global* threshold for error for the entire B-tree, called the *interpolation diameter* Δ. We call a B-tree node "*interpolation-friendly*" if any key can be found with interpolation error of at most Δ, i.e. $\Delta \geq max(MaxErr_{\text{Left}}, MaxErr_{\text{Right}})$. If a node is interpolation-friendly, we can estimate the location corresponding to the query point q, say $\hat{p_q}$, and then search the area $\left[\hat{p_q} - \Delta, \hat{p_q} + \Delta\right]$, as the result is guaranteed to be in this area. Another advantage of having a pre-defined value for Δ, is that the length of the loop for searching in a node is defined in compile time, hence the program can be efficiently optimized using efficient branch-free and/or loop-unrolled code for either linear or binary search [2].

Marking IFB-tree nodes. Building an IFB-tree involves two steps. The first step is to build a simple B-tree. In the second stage, we analyze all nodes in B-tree to find which nodes are interpolation friendly. More specifically, a node is interpolation-friendly if $|p_{\text{key}_i} - i| < \Delta, \forall\ 0 \leq i \leq K$. Once a node is identified as interpolation-friendly, it should be flagged somewhere. In case that the interpolation error for a node is above Δ, the default search procedure should be used without interpolation, which can be done by either linear or binary search.

A typical B-tree node consists of a list of keys and pointers to next nodes. However, the MSB of the 64-bit pointer is rarely used in practice, because the memory address in commodity operating systems is less than 2^{64-1} bytes = 2 Exabytes. Therefore, we can use the MSB of the pointer to flag if the node is interpolation-friendly. To use the pointer, we need to mask the MSB of the pointer value before using it. Figure 2a shows the layout of an IFB-tree node. Note that if IFB-tree is used as a primary index, the data tables are sorted too, hence interpolation can be also done on the leaf node to predict the position of the result in physical table pages. Moreover, marking an IFB-tree is a lightweight process compared to B-tree build time, but it can be easily parallelized as well.

2.4 Complexity analysis

The complexity of IFB-tree is the same as the B-tree for all operations. Whenever an IFB-tree node is created or modified, the interpolation error must be re-computed for all values in the



(a) Interpolation in IFB-tree **(b) Overall layout of IFB-tree. IF nodes are marked in red.**

Figure 2: Interpolation in IFB-tree

node. For a node with node size of p, the time taken to find the maximum interpolation error for all values in the node is $O(p)$. In the following, we analyze the time taken to build and update the IFB-tree and show that none of the theoretical complexities are different than that of B-tree.

Build time.. Bulk-loading data in a B-tree takes $O(n \log_p n)$ time. In IBF-trees, there is a second phase for evaluating the nodes and marking interpolation-friendly nodes, which takes $O(n)$. The complexity of bulk insertion thus is $O(n \log_p n + n) = O(n \log_p n)$.

Update time. For insertion and deletion, the complexity of the B-tree is $O(p \log_p n)$, which consists of $O(\log_p n)$ access/modification of nodes times $O(p)$ time for modifying each node. When using IFB-trees, we simply re-evaluate the interpolation-friendliness of each node when an element in the node is modified. The complexity of accessing a node thus still is $O(p + p) = O(p)$, leaving the overall $O(p \log_p n)$ complexity unchanged for update operations.

Lookup time. Finding the position corresponding to the query in each node takes $O(t)$ time if the node is interpolation-friendly (t=interpolation diameter), and $O(p)$ otherwise. Depending on how many of the nodes are interpolation-friendly. As $t < p$, the lookup time is between $O(t \log_p n)$ if all nodes are interpolation-friendly and $O(p \log_p n)$ in the worst case.

3 EVALUATION

In this section we compare the performance of the IFB-tree for both synthetic and real-world data to analyze its lookup time and the effectiveness of interpolation.

Experimental Setup. IFB-tree and B-tree are implemented in C++ and compiled with gcc (7.3.0). Note that all data resides in main memory. The range index finds the first the first indexed key that is equal or higher than the lookup key. Also, the keys on the physical layout are sorted (i.e. it is a clustered index), so that the entire result set can be returned once the first key is found.

Datasets. We used three datasets for performance evaluation, namely accessLog, lognormal, and longitudes. The accesslog data contains web-server log records from a hotel booking website. Lognormal is a synthetic data generated from lognormal distribution. Longitudes are sampled without replacement from the longitudes of over 1.5 billion locations extracted from the OpenStreetMap database [17].

Implementation details. For both B-tree and IFB-tree, we used 64-bit keys and 64-bit payload. Scanning in each node can be done by either linear or binary search. When the baseline methods use linear search, the performance improvements of IFB-trees are expected to be higher and more predictable, because shrinking the scan area will linearly reduce the lookup time in nodes. However, it is generally reported that binary search is more effective on new hardware, even for medium-sized nodes [2, 7]. We consequently use binary search for both IFB-tree and the baseline B-tree methods.

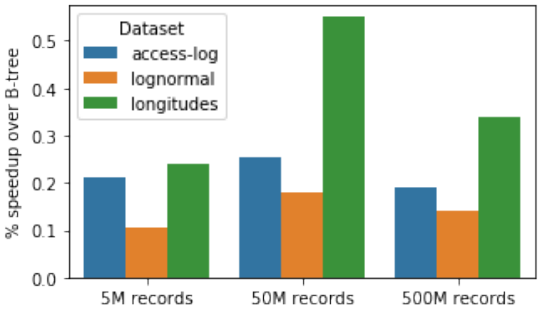


Figure 3: Speedup gained by interpolation (IFB-tree)

Speedup. Figure 3 shows the speedup obtained by the IFB-tree over the B-tree on different datasets with 5M and 500M keys. After tuning the parameters for both indexes (i.e., page size and interpolation diameter), the IFB-tree improves the performance by 10% to 55%.

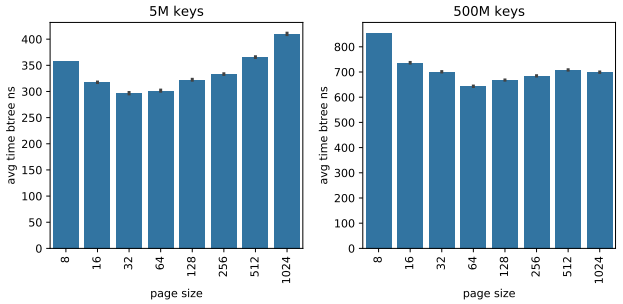
Effect of page size. The key parameter in a B-tree is the page size (= node size). Figure 4a shows how the B-tree lookup time is affected by the page size. Choosing a large page size decreases the branching factor and the tree depth, but searching through each nodes takes more time.

Speedup analysis. The performance speedup of the IFB-tree depends of B-tree nodes that are interpolation-friendly, as well as the interpolation diameter (the area that should be searched within each page). As shown in Figure 4b, the number of interpolation-friendly nodes is almost proportional to $\frac{\Delta}{\text{page size}}$, i.e., to interpolate the majority of nodes on large pages, we need a larger error window. However, a larger error window also decreases the speedup gained by interpolation. Figure 4c shows the speedup of IFB-tree against a B-tree with the same page size, suggesting that the best error window is obtained when $\Delta = \frac{1}{4} \times$ page size. The lookup times (in nanoseconds) are depicted in Figure 4d.

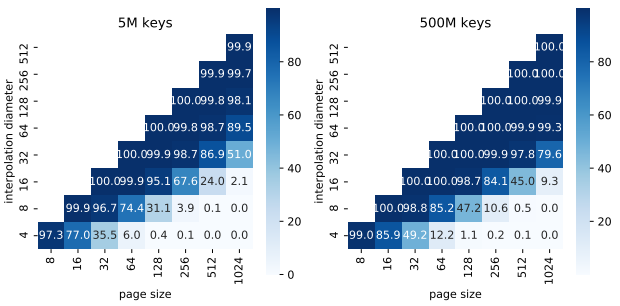
4 RELATED WORK

Interpolation search. Interpolation is an alternative to index structures for estimating the location of records in a clustered index. The technique is known to be very effective when the underlying data distribution is close to uniform [4].

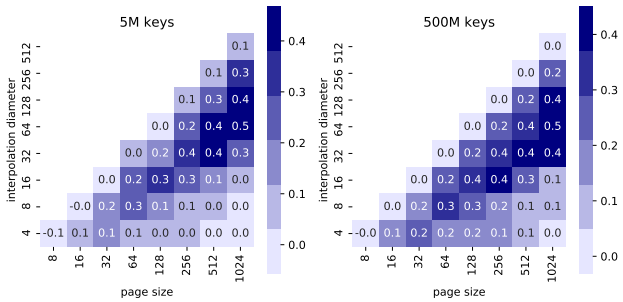
Learning database engine. The use of machine learning on learned indices is just recently proposed. Kraska et al. suggested a learned index based on a hierarchy of deep models, called the *Recursive Model Index* (RMI) [9]. Further research is done on linear learned indexes [3], inverted indexes [18], and enhanced learned bloom filters [15, 18]. Moreover, some theoretical analysis is done on the maximum *capacity* of deep learning models for



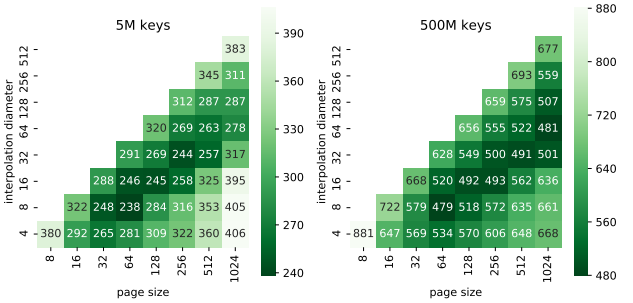
(a) Optimal page size for B-tree



(b) Percent of interpolation-friendly nodes



(c) Speedup over B-tree (with the same page size)



(d) Lookup time for IFB-tree (ns)

Figure 4: IFB-tree analysis on Longitudes dataset

keeping index key information [20]. Machine learning has been also adopted for other database operations [6, 8, 10, 13, 13, 16, 19].

Data transformation. Following a different line of work, it is suggested that the performance of index structures can be greatly improved by pre-processing the input data and transforming the keys using a mapping function, in a way that the index structure be more efficient for indexing the distribution of the transformed keys. This idea is exploited for bloom filters [14] and multi-dimensional data [1]. Interestingly, a data transformation that makes the key distribution closer to uniform can benefit IFB-tree and further decreases the interpolation error, hence boosting the speedup of an IFB-tree against its B-tree equivalent.

B-tree enhancement techniques. Several suggestions have been made to design B-trees efficient for the modern hardware. For example, Levandoski et al. suggested BW-trees, a latch-free and cache-friendly B-tree that is used in several Microsoft data engines [12]. Optimizing B-trees can involve compression techniques such as prefix-based splitting. Leis et al. suggested Adaptive Radix Trees, which consumes less memory especially on top levels of the tree. Since all tree-based data structures follow the basic principles of B-trees, such as maintain data in sorted order and splitting the distribution using hierarchies of the table, IFB-tree can be customized for these variants, too.

5 CONCLUSION AND FUTURE WORK

We suggest that the ideas behind learned indexes can be integrated with classical index structures to make them aware of the distribution, hence boosting the performance of the current algorithm. By adopting a computationally lightweight method like interpolation, we boosted the performance of B-trees by up to 50% without modifying the overall structure of B-trees or deteriorating their theoretical performance guarantees. The intra-node interpolation idea is indeed independent from the layout of the tree and hence can be integrated in different extensions to B-tree, including B+-trees, BW-trees [12], and radix trees [11].

REFERENCES

[1] Anonymous. . Spreading vectors for similarity search. In *ICLR 2019 Submission.*

[2] Performance comparison: linear vs binary search. https://dirtyhandscoding.github.io/posts/performance-comparison-linear-search-vs-binary-search.html.

[3] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2018. A-tree: A bounded approximate index structure. *arXiv preprint arXiv:1801.10207* (2018).

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

[5] Goetz Graefe et al. 2011. Modern B-tree Techniques. *Foundations and Trends in Databases* 3, 4 (2011), 203–402.

[6] Milad Hashemi, Kevin Swersky, Jamie A Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning Memory Access Patterns. *arXiv preprint arXiv:1803.02329* (2018).

[7] Tomas Karnagel, Roman Dementiev, Ravi Rajwar, Konrad Lai, Thomas Legler, Benjamin Schlegel, and Wolfgang Lehner. 2014. Improving in-memory database index performance with Intel Transactional Synchronization Extensions. In *HPCA.* 476–487.

[8] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. *arXiv preprint arXiv:1809.00677* (2018).

[9] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *SIGMOD.* 489–504.

[10] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to Optimize Join Queries With Deep Reinforcement Learning. *arXiv preprint arXiv:1808.03196* (2018).

[11] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE.* 38–49.

[12] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE.* 302–313.

[13] Ryan Marcus and Olga Papaemmanouil. 2018. Deep reinforcement learning for join order enumeration. *arXiv preprint arXiv:1803.00055* (2018).

[14] Michael Mitzenmacher. 2018. A Model for Learned Bloom Filters and Related Structures. *arXiv preprint arXiv:1802.00884* (2018).

[15] Michael Mitzenmacher. 2018. Optimizing Learned Bloom Filters by Sandwiching. In *NIPS.*

[16] Arvind Narayanan, Saurabh Verma, Eman Ramadan, Pariya Babaie, and Zhi-Li Zhang. 2018. DeepCache: A Deep Learning Based Framework For Content Caching. In *NetAI.* 48–53.

[17] OpenStreetMap on AWS. https://aws.amazon.com/public-datasets/osm.

[18] Harrie Oosterhuis, J. Shane Culpepper, and Maarten de Rijke. 2018. The Potential of Learned Index Structures for Index Compression. In *ADCS.*

[19] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. 2018. Learning State Representations for Query Optimization with Deep Reinforcement Learning. (2018).

[20] Alexandre Sablayrolles, Matthijs Douze, Cordelia Schmid, and Hervé Jégou. 2018. Deja Vu: an empirical evaluation of the memorization properties of ConvNets. *arXiv preprint arXiv:1809.06396* (2018).