START – Self-Tuning Adaptive Radix Tree

Philipp Fent*, Michael Jungmair*, Andreas Kipf, Thomas Neumann
Technische Universität München
{fent,jungmair,kipf,neumann}@in.tum.de

***Abstract*—Index structures like the Adaptive Radix Tree (ART) are a central part of in-memory database systems. However, we found that radix nodes that index a single byte are not optimal for read-heavy workloads. In this work, we introduce START, a self-tuning variant of ART that uses nodes spanning multiple key-bytes. To determine where to introduce these new node types, we propose a cost model and an optimizer. These components allow us to fine-tune an existing ART, reducing its overall height, and improving performance. As a result, START performs on average 85 % faster than a regular ART on a wide variety of read-only workloads and 45 % faster for read-mostly workloads.**

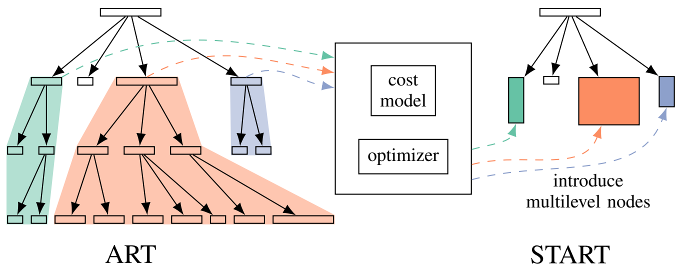

Fig. 1. Self-tuning ART results in a shallow and more efficient tree

I. Introduction

One of the go-to index structures for main-memory database systems is the Adaptive Radix Tree (ART) [13]. Several aspects make ART very favorable: It is almost as efficient to query as a hash table [1], while still storing entries in-order, which allows supporting range queries. Furthermore, it supports efficient inserts [13], which makes it overall one of the best index structures. Several fully-fledged DBMSs, including HyPer [9] and DuckDB [15], use ART in practice.

More recent research explored using machine learning to predict the location of a sought-after data item efficiently [12]. This approach showed great success in reducing the size overhead of the index while also significantly speeding up lookup times. With their more shallow structures, they often outperform ART. E.g., for data that follows a normal distribution, a machine learning approach [12] can have 2.5 $\times$ higher throughput. However, learned indexes come at the cost of robustness, where inserts or updates usually require retraining parts of the index. The efficient and robust handling of changing data is still under intensive research [7].

Especially for read-only or read-mostly benchmarks, learned indexes take the performance crown off ART [10]. We found that this is partly caused by the hierarchical structure of nodes in an ART. For the goal of accessing data in the leaf nodes, the tree structure requires data-dependent loads with pointer chasing, proportional to the height of the tree. Piecewise linear functions predict the data position reasonably good, such that accessing data may only require a single cache miss.

In this work, we introduce **S**elf-**T**uning **ART** (START), an ART-variant that features read-optimized multilevel nodes. With these new node types, START is efficient to query while still supporting fast modifications. When too many updates occur in multilevel nodes, it falls back to regular ART nodes. To simplify reasoning in this paper, we assume that we tune the tree in offline mode, but the concept can also be applied online. To introduce multilevel nodes, we take an existing ART, determine the most suitable subtrees, and transform those. Figure 1 shows this analysis and transformation process.

*Equal contribution

Our self-tuning approach deduces its configuration from first principles: We start by collecting ART metrics (i.e., the time to traverse a node) in an isolated experiment on the target machine. These measurements build the foundation of a cost model of tree lookups. Based on this model, we determine sets of (combined) nodes where the cost for a lookup in this subtree would be smaller with multilevel nodes: $C_{\text{START}} < C_{\text{ART}}$. We compute the optimal subtrees in $\mathcal{O}(n)$ using a dynamic programming algorithm that maintains the optimal subtrees for each node.

This approach makes no assumptions about workloads or underlying hardware. In preliminary experiments, we noticed that some node optimizations are very hardware-specific and might not be equally beneficial on different platforms. Instead, START relies on measurements directly on the target machine (cf. Section V), which always tunes START for the specific hardware. Besides, using a custom, workload-specific cost function $C(n)$, it could be tuned further to adapt to a particular workload.

Our contributions include:

1) Two ART node designs that accelerate data access by reducing the tree height. One design proposes a novel and compact multilevel ART node based on a technique called rewiring (Section III).
2) A model for reasoning about the costs to traverse an ART (Section IV-A).
3) An algorithm that, based on the cost model, optimally places multilevel nodes in an existing tree (Section IV-B).

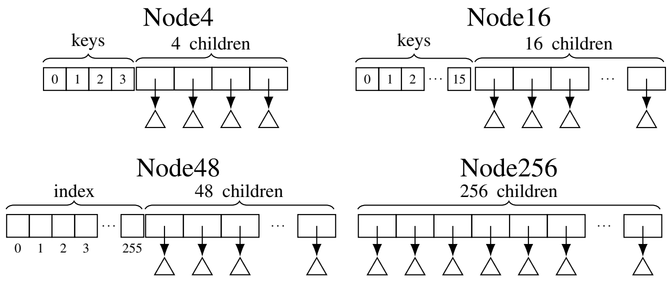

Fig. 2. Regular ART nodes that each index a single byte

II. Background: Adaptive Radix Tree

ART is a radix tree with adaptive nodes that scale with the density of values populating the key space. As shown in Figure 2, an ART can have four different node types that fall into three structural classes. Node4 and Node16 use a key array where the key position corresponds to the correct child pointer. A Node48 instead stores 256 single-byte child indexes, to which the radix key byte serves as an index. This child index then points to the corresponding child pointer. Node256 follows the regular radix tree design of storing child pointers in a flat array.

To optimize for sparsely populated regions, ART uses path compression. It allows for a significant reduction in tree height and space consumption by removing all inner nodes that only have a single child. Path compression especially helps with string keys, which would otherwise produce deep trees, but also applies to other key types. E.g., dense surrogate integer keys, which auto-increment for each element, might use 8 Byte storage, but in practice, the leading 3 Byte are 0 for up to a trillion entries.

In this paper, we argue that this static single-byte radix can be improved. In several workloads, an ART is higher than necessary, which causes many data-dependent random accesses. However, for practical considerations, it still makes sense to use a byte-wise radix, since many standard keys naturally follow byte boundaries. In the following, we introduce new ART nodes that span multiple key-bytes and levels.

III. Multilevel Nodes

In contrast to regular ART nodes, multilevel nodes allow access with multiple key-bytes. Since this necessarily makes the nodes span more entries than traditional ART nodes, this is most beneficial for read-heavy workloads. By introducing larger and more complex nodes, START reduces the height of the tree and trades off insert for lookup performance. Still, compared to learned indexes, START can be efficiently updated without retraining learned models.

Without formal proof, we argue that the multilevel nodes that we introduce do not change the worst-case space consumption of 52 Bytes per key-byte. The intuition is that we can extend the original ART proof [13] by keeping the space consumption $s(n)$ of a multilevel node within the combined budgets $b(x)$ of all child nodes $c(n)$:

$$s(n) \overset{!}{\leq} \left(\sum\nolimits_{x \in c(x)} b(x)\right) - 52$$

In other words, we can guarantee the worst-case space consumption by keeping multilevel nodes small enough. That is, we keep the size of START's multilevel nodes within the budget of its direct children.

A. *Rewired Nodes*

Naïve multilevel nodes, similar to Node256, could use flat arrays for 2-level and 3-level nodes, holding $256^2 = 64\,\mathrm{K}$ and $256^3 = 16\,\mathrm{M}$ entries. Indexing into such nodes would be similar to Node256 by interpreting 2, respectively 3 Bytes as an index into an array of child pointers. However, this implementation is too inflexible and space inefficient for anything other than very dense regions in the key space.

Instead, we propose two new multilevel node types that make use of rewired memory [16] to obtain a more compact representation for sparser areas. Rewiring decouples the logically indexable virtual address space from physical pages that consume memory.

We follow the implementation of rewiring presented in [16]: First, we create a main-memory file backed by physical pages using the Linux system call `memfd_create`. Then, we map this file page-wise to virtual memory using multiple `mmap` calls. This procedure is depicted in Figure 3: A total of four virtual pages are mapped to the created main-memory file. The pages of the main-memory file are then backed by physical pages on demand.

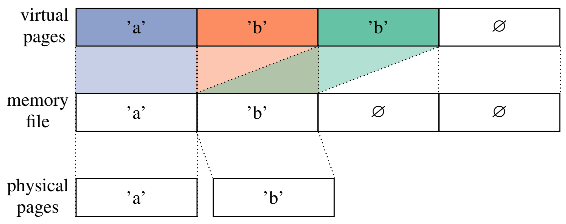

Fig. 3. Implementation details of rewiring

In the example, the first two virtual pages (blue and red) are mapped to the corresponding two pages of the main-memory file. Using a second `mmap` call, the green virtual page is also mapped to the second file page. Thus, both the red and green virtual pages are mapped to the same physical page with content 'b'. The white virtual page is mapped to the file, but is not backed by physical memory since it has not been accessed.

We use rewiring as shown in Figure 4: multiple compatible and sparsely populated virtual pages can share one physical page. Additionally, empty pages do not consume any physical memory.

In the example, we set up the first virtual page as regular memory with a direct mapping to physical memory. In contrast, the following two virtual pages share one underlying physical page. As a result, the memory contents of these pages are now from the same physical page and need another distinguishing

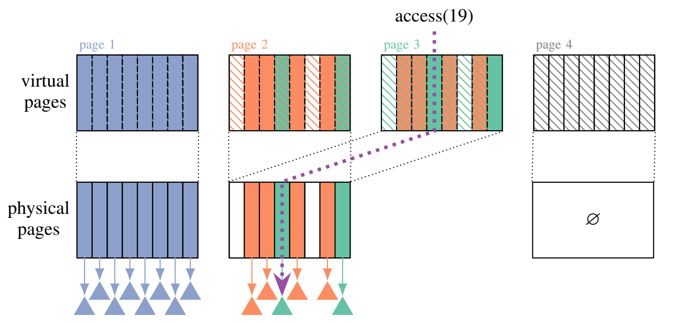

Fig. 4. Implementation of page-based rewired nodes

factor. We use pointer-tagging (denoted in color) to detect empty slots that are used by another virtual page. E.g., on page 2, the fourth entry is occupied by an element of page 3, which we indicate by different colored hatching and fill. Thus, when accessing an entry, we check that the tag (solid color) matches the page number (hatching). Empty nodes are left mapped to the zero page "∅", not consuming any memory until the first write.

We observed the effectiveness of this rewiring on a real-world dataset (`osm_cellids`, cf. Section VI). On average, rewiring a Node16M (256^3 child pointers on 2^{15} virtual pages) already pays off for less than 200 k entries (a fill rate of about 1 %). Such a node only consumes 22.5 Bytes per entry. As a result, the space overhead is comparable to that of a Node256, which needs a fill rate of at least 19 %.

However, page rewiring also has some limitations. Since we set up pages individually, instead of amortized through a memory allocator, we need to execute many costly `mmap` calls, which again cause expensive page faults. Furthermore, page sharing only works for "well-behaved" workloads, that have few collisions on child pointers. Inserts may even cause new collisions, which we handle with explicit unsharing operations during insertion.

B. *Multilevel Node4*

Another observation that we made is that path compression already stops working when there are two keys in the path. Multiple keys lead to an excessive number of nodes in this path, which can degrade performance due to chains of Node4. Our idea is to use the 12 unused Bytes of a regular Node4 (with respect to its representing cache line) to add multilevel support. The resulting multilevel Node4 can store up to four keys with up to four bytes each while not exceeding the size of a cache line (64 Bytes).

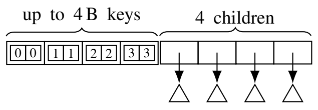

Fig. 5. Multilevel Node4 can store multiple key bytes in a single cache line

As depicted in Figure 5, we keep the layout of the regular Node4 type. Following the node header, we still store up to four keys that now can have a size of up to four bytes. The only difference is that now key parts have a size of up to four bytes, and the Node can, therefore, span over up to four radix levels.

IV. Self Tuning

A key question is where and when to use multilevel nodes in an ART. In regular operation, we have only node-local information without the full context of the surrounding. It would be too expensive to collect that information during inserts, and storing it would significantly increase memory usage. Instead, we introduce a self-tuning phase that computes the context once for the whole tree and optimally places multilevel nodes. Similar to retraining a learned index, the cost of this phase can be amortized over time. E.g. it only runs when more than 10 % of the data was changed, or it runs as part of regular index maintenance (e.g., during `VACUUM` jobs).

Even with global information, it is still not obvious where to optimally place multilevel nodes. While the replacement of a subtree with a multilevel node might be beneficial, it might prohibit another more optimal configuration. To solve this problem, we first introduce a cost model. Then, we present a dynamic programming algorithm that uses the cost model to minimize the overall lookup costs.

A. *Cost Model*

To obtain the cost for a lookup in a node, we measure the cost $C_L(n)$ to traverse a node n in an isolated experiment. As baseline $C_L(\varnothing)$, we take the time to traverse two consecutive Node4. Then, we measure the time $C'_L(n)$ to traverse two Node4 with the node n in between these two nodes, which gives $C_L(n) = C'_L(n) - C_L(\varnothing)$. To get stable results, we take the median of multiple runs, separated by an `LFENCE` instruction as a speculation barrier. We found that a less involved experimental setup does not model the costs of inner nodes as well, which is due to speculative execution and caching effects.

Another factor for traversal costs is that, in general, upper levels of the tree are hotter than lower levels. Upper-level nodes usually reside in cache, while lower layers are cold. Therefore, we also measure the costs to traverse cold nodes by explicitly clearing the cache for these nodes using `CLFLUSH`. Since caches are transparent, costs can only be measured reliably for nodes residing in L1 cache or in main memory: $C_{L,L1}(n)$ and $C_{L,RAM}(n)$.

In practice, we may want to extend this to consider more cache levels. For given latencies of caches $L_{L1}, L_{L2}, \ldots$ and main memory L_{RAM}, we can derive approximate costs $C_{L,Li}(n)$ for a lookup of a node n residing in the i-th cache level:

$$C_{L,Li}(n) = C_{L,L1}(n) + \left\lfloor \frac{C_{L,RAM}(n)}{L_{RAM}} \right\rfloor \cdot (L_{Li} - L_{L1})$$

This formula extrapolates the costs for nodes in lower cache levels (L2 and L3) based on known cache latencies and actual node traversal costs measured for L1 cache. For this purpose, we estimate the number of accessed cache lines by dividing the lookup cost for uncached nodes by the main memory latency.

For the definition of the cost model, we first define two auxiliary functions: $\#_k(n)$ denotes the number of keys stored in the subtree underneath n. $\text{child}^+(n,d)$ is the set of all nodes with a distance of exactly d child pointers from n. For example, $\text{child}^+(n,1)$ is the set of all direct children of n. Based on these functions, we define the cost $C(n)$ for a regular ART node as the measured cost and the cost of accessing all keys in the subtree underneath n:

$$C(n) = C_L(n) \cdot \#_k(n) + \sum\nolimits_{x \in \text{child}^+(n,1)} C(x)$$

In this formula, the cost of a node n is the cost of traversing it on its own weighted by the number of keys that can be accessed through n. Plus, recursively, the cost of all nodes in its subtree. If we instead use a multilevel node o spanning i levels, we skip the next $i-1$ levels:

$$C_{Replaced}(o,i) = C_L(o) \cdot \#_k(o) + \sum\nolimits_{x \in \text{child}^+(o,i)} C(x)$$

We thus define the benefit b of replacing node n with the i-multilevel node o as follows:

$$b(n,o,i) = C(n) - C_{Replaced}(o,i)$$

We want to point out that this cost model is workload-agnostic, i.e. it allows to minimize the *average* cost of visiting keys in the tree. A workload-aware cost model could help START to adapt to actual query patterns.

B. Optimization

A first approximation of optimal node placement would be a greedy bottom-up replacement. Since lower-level nodes are less likely to be cached, thus have higher average cost, replacing the bottom-most nodes with multilevel nodes results in surprisingly good results. However, this can produce suboptimal trees when mid-level nodes would be even more beneficial.

Instead, we propose a dynamic programming algorithm [2] that considers the cost of optimal lower-level nodes. We first consider the optimal costs of its descendant subtrees to determine if we want to introduce a new i-level node. We denote the intermediate results next to individual nodes as follows.

$$\begin{gathered} 40 \\ [295, 215, 135, \ldots] \end{gathered}$$

The first line denotes the number of keys of the subtree (the sum of the number of keys of the direct children). The second line contains a list of combined optimal costs of its descendants. We start with the costs for the 0th descendant, i.e., the subtree itself, then the 1st descendants, i.e., the sum of the costs of its children, and we continue with the remaining descendants. Hence, there are at most as many entries as there are levels under a certain node. We bound the length of this array to the deepest multilevel node that we support.

Figure 6 shows an example calculation in a 2-level subtree. We traverse the tree bottom-up, starting with the bottom-most layer, reusing already computed aggregates. In the mid-level layer, we do not consider replacements and just compute the cost table of a single-level node with cost 2. To calculate the optimal subtree costs, we fill the array backward. We start with the sum of costs of the 1st descendants: $80 = 60 + 20$. Then

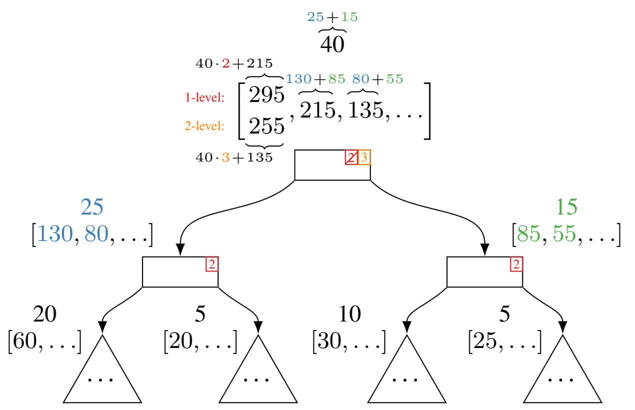

Fig. 6. Example for minimizing subtree costs using dynamic programming

TABLE I
EXAMPLE COSTS FOR A LOOKUP IN A NODE

[ns/lookup]	Levels	Cached	Header Cached	Uncached
Node4	1	7	7	68
Node16	1	5	77	162
Node48	1	2	165	168
Node256	1	2	88	92
Rewired64K	2	6	87	162
Rewired16M	3	6	88	165
MultiNode4	2-4	6	6	68

we compute the cost of the subtree itself as the number of keys times the cost plus the (already calculated) cost to traverse the lower descendants: $130 = 25 \cdot 2 + 80$.

For the top-most layer, we consider a single-level node with cost 2 (red) and alternatively a 2-level node with cost 3 (orange). The calculation for the single-level node is analogous, but the 2-level computation combines the node's cost with the cost of its 2nd descendants instead ($135 = 80 + 55$). In the example, the resulting optimal subtree uses the 2-level node, outperforming the optimal subtrees of the lower levels.

Once the bottom-up calculation reaches the root, it serves as a blueprint to introduce any new, optimally placed multilevel nodes. Since upper-layer decisions during the first traversal might cause optimal lower-level node creation to be skipped (because they are incorporated into multilevel nodes), we traverse the tree a second time to transform nodes in bulk. For this transformation step, we attach multiple helper structures to the tree. Afterward, we discard all helper structures, which leaves zero overhead for regular operation.

V. Node Evaluation

In the following, we present cost model measurements, as described in Section IV-A, for regular as well as multilevel nodes. All measurements were executed on an Intel i9-7900X CPU with 13.8 MByte last-level cache (LLC) on a single NUMA node. As a point of reference, an LLC miss takes about 60 ns.

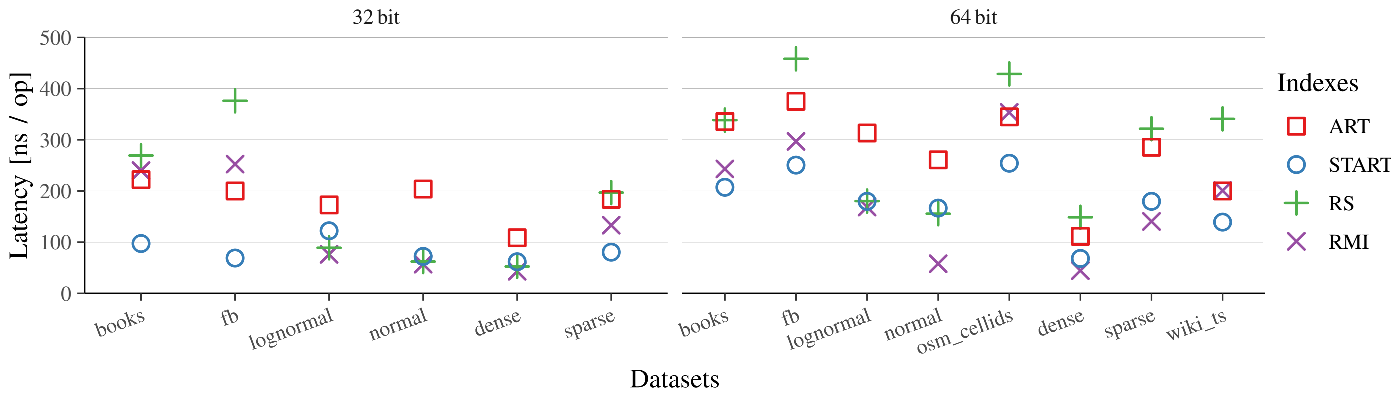

Fig. 7. Read-only performance of a regular ART and START in comparison to RMIs and RadixSplines

Table I shows a performance evaluation of the individual nodes. For each node type, we measure lookup times for three different cases:

Cached: the node resides completely in L1 cache
Header Cached: only the first cache line of the node is in L1 cache, while the rest of the node is flushed
Uncached: the whole node is not in the cache

We measure these three different setups since the header is the most accessed part of the node. A traversal unconditionally accesses the header to determine the node type, while it only needs a single child pointer. For many node types, this causes one additional cache miss, but especially for larger nodes, the header is hot while the body is relatively cold.

The results of the first two nodes Node4 and Node16 are as expected: A Node4 fits into one cache line, and therefore a cold access causes one cache miss. Node16 needs to access two cache lines, as both the key array and 16 child pointers do not fit into 64 Byte. On the other hand, the results for Node48 and Node256 are surprising at first: We would expect a total of three cache misses for Node48 and two for Node256. However, our measurements show significantly less time spent during lookup, corresponding to one fewer cache miss than expected. This can be explained with correct branch prediction and out-of-order execution of modern processors. The processor correctly predicts the node type and speculatively executes its lookup code, effectively hiding the second cache miss. Since both Node48 and Node256 do not use the header for a lookup, we can only observe the latencies of two and one cache miss(es), respectively.

In our evaluation setup, naïve multilevel nodes Node64K and Node16M behave identical to Node256, again exploiting out-of-order execution. Still, these implementations are impractical due to their memory consumption and inflexibility. In situations where at least the header is cached, our implementation of rewired nodes shows nearly identical performance. Due to an implementation detail of rewiring, we keep the header separate from the rewired structure and store the start address of the rewired child pointer array. This leads to two observable cache misses if the node is cold, but due to the sheer size of the nodes, their header is usually very hot and, therefore, cached. Then only a single cache miss occurs for accessing the entry. MultiNode4 is also a straight improvement to Node4, spanning multiple levels with almost the same performance.

In summary, START's multilevel nodes can offer significantly lower lookup times. E.g., we can get much lower latency if we replace three layers of Node256 by one Rewired16M node where one single header can still fit into the cache:

$$88 + 2 \cdot 92 = 272 > 88_{\text{Rewired16M}}$$

VI. Overall Evaluation

In the last section, we noted that *individual* multilevel nodes can be significantly faster than a combination of regular nodes. However, the global impact of multilevel nodes is not immediately apparent. In the following, we present an overall evaluation, primarily based on SOSD [10], introduced initially as a benchmark for learned indexes. SOSD provides a reliable single-threaded benchmarking setup for a multitude of datasets and comes with many (learned) index structure baselines.

Each SOSD dataset consists of 200 million unsigned integer keys, either 32 bit or 64 bit. Some datasets are taken from real-world data (`books`, `fb`, `osm_cellids`, `wiki_ts`), while others are synthetic (e.g. according to a lognormal distribution). In our evaluation, we compare with the following four index structures:

ART: A regular adaptive radix tree
START: Our C++17 implementation of a self-tuning ART[1]
RS: RadixSpline (cf. Section VII-B)
RMI: Recursive Model Index (cf. Section VII-B)

A. Read Only

The SOSD benchmark performs read-only measurements. It records the time needed to perform ten million lookup operations on the index structure (with keys drawn from the dataset). Figure 7 shows the resulting average lookup latency of the four index structures for all 14 datasets.

We observe that START is *consistently* faster than ART, over 85 % on average over all datasets. For the real-world `fb` dataset, the lookup latency of START is even three times lower than ART's. Moreover, START is also competitive with the two learned indexes. For all real-world datasets, START has

[1]https://github.com/jungmair/START

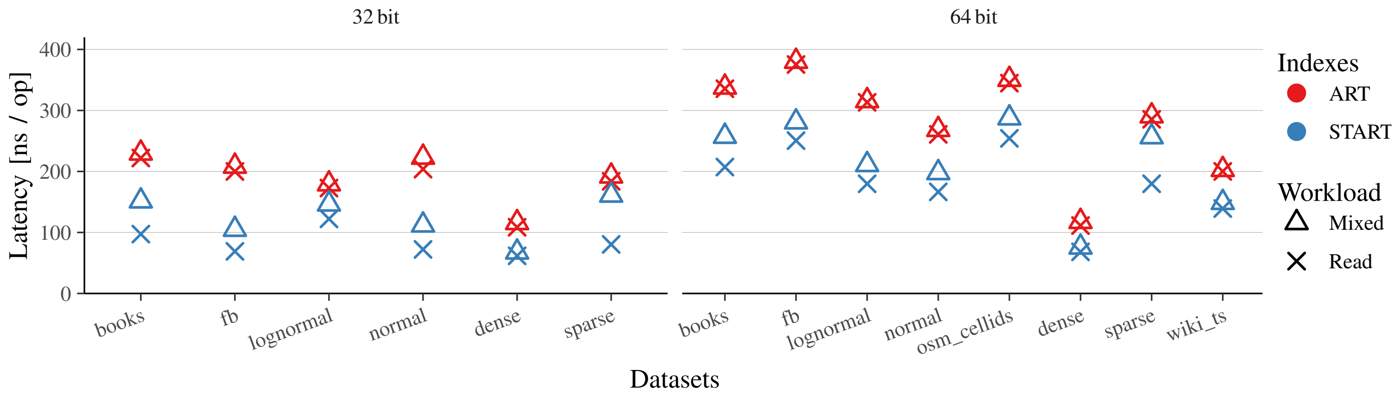

Fig. 8. Mixed read and write workload performance of a regular and a self-tuning ART

better performance than both RadixSpline and RMI. Even for synthetic datasets, START is competitive with learned indexes.

B. Read Mostly

In contrast to the learned indexes, START can not only be used as a read-only index but also supports efficient updates. We, therefore, additionally run a benchmark with writes. Since our multilevel nodes are specially optimized for read-heavy workloads, START's insert operations are more expensive but still feasible: For a write-only workload, START is, on average, slower by 71 % than ART. To create a more realistic read-mostly workload, we use a benchmark akin to YCSB-B [6]. In this benchmark, we split the workload into 95 % reads and 5 % skewed writes. The skew follows the YCSB "Latest" distribution, which is a Zipfian [8] distribution, where higher keys are more likely to be chosen.

From the original dataset of 200 million keys, we sample 500 thousand keys according to this distribution and exclude them during the build phase. Then, we measure the time to insert these keys into the index and subsequently perform 9.5 million lookups. Since inserts might cause the multilevel nodes to degrade, we consider this the worst case for START.

Figure 8 shows results for ART and START in this mixed workload compared to read-only. Since the learned indexes do not support inserts, we exclude them from this experiment. As expected, performance degrades with concurrent insert operations. Still, START is over 45 % faster than a regular ART and is even better in mixed operation than an ART in the read-only workload.

C. Memory Consumption

It is important to note that while START's memory consumption is guaranteed to stay in the same theoretic bounds than ART, we do not optimize for space consumption but for lookup time. Still, on average over all datasets, START consumes a similar amount of memory compared to ART. Yet, memory consumption depends on the dataset: For e.g., the 64-bit `books` dataset, START requires 8.3 GB of RAM instead of 5.4 GB for ART. On the contrary, for the 64-bit `normal` dataset START uses only half of the memory allocated by ART: 2.1 GB are used compared to 4.7 GB allocated by ART. The effectiveness of shared physical pages varies depending on the dataset which explains the differences in memory consumption.

VII. Related Work

Index structures are well researched, with a multitude of different approaches to quickly find the values for a given key. We present related work that broadly falls in the two categories of traditional and learned index structures.

A. Traditional index structures

The Adaptive Radix Tree has been thoroughly analyzed, compared with different data structures, and received proposals to improve it [1], [13], [18]. Wong et al. [17] analyze the performance of ART regarding the Translation Lookaside Buffer (TLB). They find that a regular ART can have performance problems caused by TLB misses and propose a workload-conscious reorganization of nodes onto "hot" pages. This reorganization step to reallocate nodes and introduce huge pages is similar to our analysis step but does not introduce different node types.

The KISS-Tree [11] is an advancement of the Generalized Prefix Tree [5] and is a very efficient radix tree with only three levels. It stores 32 bit keys using an open addressing scheme for the first 16 bits, relying on virtual memory to save space. Similar to Section III-A, completely unused ranges in the key space are only mapped to physical memory on-demand and do not use memory until used. In contrast to our rewired nodes, where pages can be shared, even a single entry in the range still uses a whole 4 KByte page of memory.

The Height Optimized Trie [4] uses a dynamic number of bits for the fanout of each level. Especially for sparsely distributed and string-like keys, this can reduce the tree height, reduce memory consumption, and increase throughput performance. In comparison to ART and our work, their finer and more granular bit-level adaptive node sizes allow an even better adaption to sparse key distributions. However, HOT has a lower maximum fanout than ART (32 instead of 256). Thus, HOT does not adapt to relatively dense areas. In contrast, START increases ARTs maximum fanout to 16M.

B. Learned indexes

The category of learned indexes introduces machine learning models as a component of index structures. Kraska et al. [12]

describe recursive-model indexes (RMI) that use staged models to predict the position of the data in a sorted data array. RMIs approximate the underlying CDF of the keys and use that model to predict the position of a key, which is followed by a local search. They show that these structures feature significantly faster lookups than traditional indexes. Nevertheless, several critical aspects for robust real-world usage like updates, concurrency, or recovery are not yet well understood.

Another approach is RadixSpline, which has a similar idea to RMIs by fitting an approximate function to the CDF [10], [14]. In contrast to RMIs that are built top-down, a RadixSpline is built bottom-up. A RadixSpline fits a linear spline to the data and uses a radix lookup table to locate the corresponding spline segment.

To solve the question of dynamic updates, Ding et al. [7] propose ALEX as an extension to RMIs that keep the underlying data in efficiently updatable arrays [3]. They propose to make learned indexes practical by using adaptive RMIs where nodes can be split on inserts. Surprisingly, the updatable arrays can also improve the lookup performance since the empty space used for efficient updates can help to keep records close to their predicted position.

VIII. Conclusion

In this work, we have shown that multilevel nodes can significantly improve the performance of ART. As we have shown, it is possible to introduce a node-by-node cost model that we successfully use for optimal node placement in START. For read-heavy workloads, we bring START close to the performance of learned indexes while retaining the robustness of ART.

Acknowledgments: This project received funding by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 725286).

References

[1] V. Alvarez, S. Richter, X. Chen, and J. Dittrich. A comparison of adaptive radix trees and hash tables. In *ICDE*, 2015.
[2] R. Bellman. On the theory of dynamic programming. *PNAS*, 1952.
[3] M. A. Bender and H. Hu. An adaptive packed-memory array. In *PoDS*, 2006.
[4] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis. HOT: A height optimized trie index for main-memory database systems. In *SIGMOD*, 2018.
[5] M. Böhm, B. Schlegel, P. B. Volk, U. Fischer, D. Habich, and W. Lehner. Efficient in-memory indexing with generalized prefix trees. In *BTW*, 2011.
[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.
[7] J. Ding, U. F. Minhas, H. Zhang, Y. Li, C. Wang, B. Chandramouli, J. Gehrke, D. Kossmann, and D. B. Lomet. ALEX: an updatable adaptive learned index. *CoRR*, abs/1905.08898, 2019.
[8] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD*, 1994.
[9] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.
[10] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. SOSD: A benchmark for learned indexes. *CoRR*, abs/1911.13014, 2019.
[11] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner. *KISS-Tree*: smart latch-free in-memory indexing on modern architectures. In *DaMoN*, 2012.
[12] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *SIGMOD*, 2018.
[13] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*, 2013.
[14] T. Neumann and S. Michel. Smooth interpolating histograms with error guarantees. In *BNCOD*, 2008.
[15] M. Raasveldt and H. Mühleisen. DuckDB: an embeddable analytical database. In *SIGMOD*, 2019.
[16] F. M. Schuhknecht, J. Dittrich, and A. Sharma. RUMA has it: Rewired user-space memory access is possible! *PVLDB*, 2016.
[17] P. Wong, Z. Feng, W. Xu, E. Lo, and B. Kao. TLB misses: The missing issue of adaptive radix tree? In *DaMoN*, 2015.
[18] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen. Reducing the storage overhead of main-memory OLTP databases with hybrid indexes. In *SIGMOD*, 2016.