State of the ART: The Adaptive Radix Tree Index for Main-Memory Databases

JONAS FRITSCH, Technische Universtität München

With recent trends in hardware, main memory capacities have grown to an extent where most traditional DBMS can fit entirely into memory. This change introduced a new shift of the performance bottleneck from disk-based I/O to main memory access.

While previous index structures like the B-Tree were optimized for minimizing disk access, the adaptive radix tree (ART) is a trie-based index structure designed explicitly for in-memory usage. It utilizes newer architecture features like SIMD and caching effectively and compresses its structure dynamically, both horizontal and vertical. With these measures, ART achieves a performance that beats other state-of-the-art order-preserving indexes in both insertion and single-lookup time while having a lower memory footprint at the same time.

1 INTRODUCTION

The architecture of DBMS has constantly been evolving due to advances in hardware. Over the last few decades, main memory capacities increased from several megabytes up to thousands of gigabytes, such that nowadays, databases can fit entirely into main memory. This change significantly impacted the general architecture of DBMS, which resulted in performance improvements by several factors [5], [13].

The design of index structures used to query a set of data more efficiently was heavily influenced by the main performance bottleneck of disk I/O in traditional disk-based DBMS. Original indexes like the B-Tree designed to minimize disk accesses perform poorly in an in-memory environment. The T-Tree [7] was one of the first index structures proposed for main memory DBMS. However, over the last 35 years, the hardware landscape changed dramatically, causing T-Trees and all other index structures not explicitly designed with caching effects in mind to be rendered inefficient for in-memory usage[10]. Further focus on developing cache-sensitive index structures resulted in many different search tree variants.

Cache-sensitive search trees (CSS-Trees) [10], while utilizing cache lines efficiently, introduce a significant overhead for updates as the tree is compactly stored in an array. Similarly, the more recent k-ary search tree [12] and the Fast Architecture Sensitive Tree (FAST) [6] both utilize Single Instruction Multiple Data (SIMD) instructions for data-level parallelism (DLP) to increase performance. However, as static data structures, they do not support incremental updates. A way to circumvent this limitation is to use a delta mechanism where another data structure stores differences and is periodically merged into the static structure. However, this comes at an additional performance cost. The cache-conscious B^+-Tree (CSB^+) [11] introduced as a variant of B^+-Trees improves cache utilization by reducing the need to store all different child pointers in each node.

Hash-Tables have been a popular indexing choice for main memory databases as they provide optimal $O(1)$ - as opposed to $O(\log n)$ for search trees - single-lookup and update time on average. Many different hashing schemes and hash functions have been developed over time, but hash-tables generally do not support any range-based queries due to hash functions scattering keys. Additionally, hash-tables can require complete re-hashing with $O(n)$ complexity upon reaching their load balance.

Author's address: Jonas Fritsch, Technische Universtität München, Fakultät für Informatik, fritsch@in.tum.de.

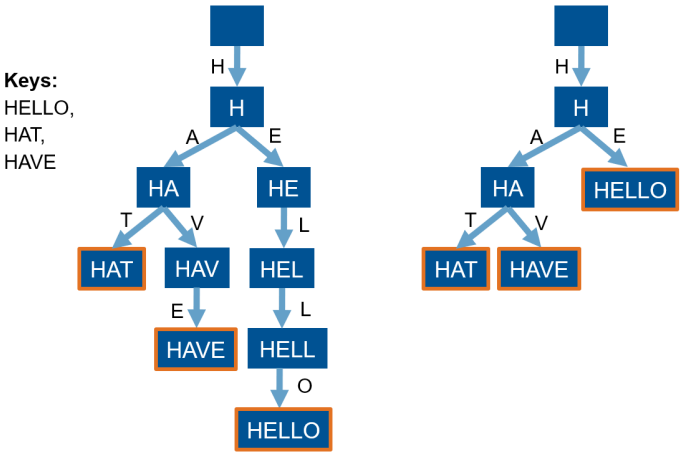

Fig. 1. A trie with span 8 (1 byte per character) storing the keys "HELLO", "HAT", and "HAVE" on the left and its radix tree variant on the right. Nodes marking the end of a word are outlined.

Another possible data structure usable for indexing is a trie. Tries are search trees with the difference that keys are inserted in pieces (partial keys). This means that two keys sharing a common prefix will have the same path from the root until their first partial key differs (common prefix compression). A radix tree, sometimes also called Patricia-Trie, is a variant of a trie that further compactifies its structure by compressing nodes for partial keys that only have one child, as shown in Figure 1 (unique postfix compression). With radix trees, some form of leaf nodes holding the compressed rest key is required to restore full keys.

Tries have the following interesting properties:

- Tree height and complexity do not depend on the number of keys stored n but rather the key length k. As we will see later in Section 4, this means that its performance is mainly impacted not by the amount of keys present as in other search trees but by the skewness of those keys.
- Tries require no rebalancing.
- All insertion orders result in the same tree.
- Keys are stored in lexicographic order.
- Keys are stored implicitly along paths. (This is not directly true for radix trees and can differ with implementations.)

The span of a trie is the number of bits making one partial key. The fanout of a node is the number of children a node can have maximum. The simplest implementation for a trie of span s is to

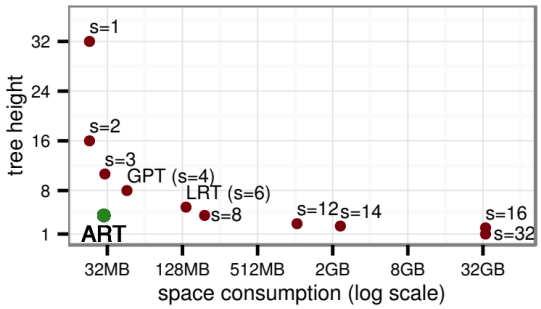

Fig. 2. Comparison of tree height versus space consumption for different radix tree spans s when storing 1M uniformly distributed 32 bit integers with 8 byte pointers. Source: [8]

have a fanout of 2^s on each node. This means that when storing the children in a pointer array of size 2^s, the partial key can be used directly as index into this array to find the next child without having to make any comparisons. With this, the height of a given trie storing keys of k bits is bound by $\lceil k/s \rceil$, and increasing the span results in a lower trie height which is desirable as the height dictates the time complexity for almost all operations.

On the other hand, increasing the span results in an exponentially higher fanout, thus requiring more space as even if few children exist, the array will be filled with null pointers wasting memory. For this reason, having a very high span is generally impractical. The Generalized Prefix Tree (GPT) [2] has a span of 4, and the radix tree implementation used in the Linux kernel (LRT) uses 6 bits [3]. The Adaptive Radix Tree (ART) [8], as depicted in Figure 3, while using a span of 8, manages to have both fairly low memory consumption as well as a small tree height.

The key idea in lowering memory consumption while maintaining a high span for tries is to adaptively use nodes with different fanouts based on the number of actual children present. One of the first data structures to utilize such adaptive nodes was the Judy Array [1] which was invented by Doug Baskins and developed at Hewlett-Packard research labs as a general associative array. However, due to its patent and overall complexity, it has not yet been researched as much. Also, its original, almost two-decade-old design has flaws like assuming 16 byte cache-line sizes and not utilizing SIMD.

The rest of this paper is organized as follows. We first introduce the Adaptive Radix Tree as an index structure and explain its key design features. In Section 3, we show how to convert different key types to binary-comparable keys to index in ART. Section 4 evaluates ART by comparing its performance and memory consumption first in micro-benchmarks and then the TPC-C benchmark. Section 5 discusses related work and research done since the original proposal of ART back in 2013. The final part draws a conclusion and discusses future work.

2 THE ADAPTIVE RADIX TREE

The Adaptive Radix Tree (ART) was first proposed by Leis et al. [8] as a performant, order-preserving in-memory index structure. Being an adaptive radix tree similar to Judy Arrays it uses multiple node types with different fanouts to drastically lower memory consumption and

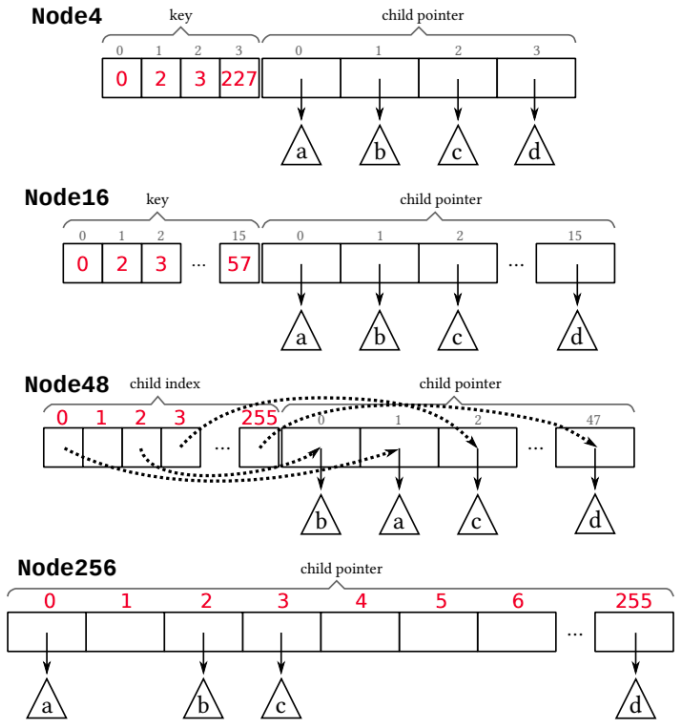

Fig. 3. Different node structures used by ART. For each node, the partial keys 0, 2, 3, and 255 are mapped to the subtrees a, b, c, and d, respectively. Source: [8]

vertical compression to reduce the overall tree height and further memory usage while improving performance.

2.1 Horizontal Compression (Adaptive Nodes)

ART uses four different node types, each with a different fanout but the same span of 8, to adaptively change between them based on the actual number of child nodes. This way, less space is wasted storing null pointers. Additionally, storing the children in sorted order allows for range scans along nodes. The node types illustrated in Figure **??** are named after the maximum number of children they can store. For better cache line utilization during searches ART stores partial keys and child pointers in two seperate arrays as opposed to storing key-pointer pairs in one single array:

Node4: Stores up to 4 children by maintaining a sorted array with size 4 of partial keys where the searched index in the key array is also the index into the child array for the associated child.

Node16: This node stores up to 16 children in the same way as Node4. Searching for a key can be done efficiently using SIMD instructions.

Node48: As the number of partial keys to differentiate increases, searches become more expensive, even in sorted order. Therefore Node48 uses a full 256-sized byte array to hold all possible partial keys. However, as this node only stores up to 48 children, our child array is only of size 48. ART thus stores the index into the child

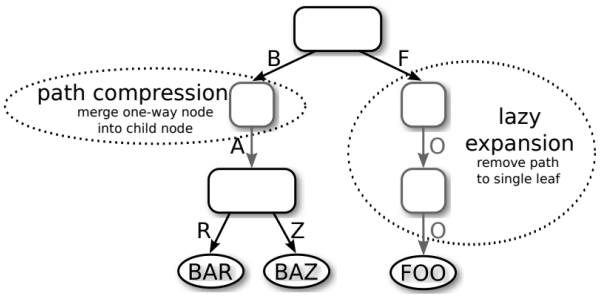

Fig. 4. Lazy expansion and path compression in effect Source: [8]

array as value in the key array, which is indexed directly using the partial key.

Node256: Finally, the largest node stores children in the classical trie approach with a child array of size 256 so that, similarly to Node48, the partial key is used directly as index.

Additionally, all nodes have a header of constant size storing the node type, the number of non-null children, and a prefix variable containing information about the compressed path (see Section 2.2).

2.2 Vertical Compression

As a radix tree, ART implements two techniques often used in tree-like structures: lazy expansion and path compression. With these ART compresses node chains where each node only has a single child, further reducing tree height and space consumption while improving performance.

Lazy Expansion: Lazy expansion refers to the standard radix tree approach introduced by Morrison [9] to expand nodes lazily upon insertion. New nodes are therefore only created as long as they are needed to distinguish between at l east two different keys. This can be seen again in Figure 4 where the key "FOO" is lazy expanded, meaning the path for storing the two "O"s is omitted. Since paths to leaves may be compressed, this requires the full key to be stored alongside its value or be retrievable based on it. The latter is most often the case in database indexes, where the value is used as a reference to a data tuple also containing the key.

Path Compression: Similarly, path compression removes nodes containing only one child and gets applied during insertion when expanding upon a previously lazy expanded key or deletion. Figure 4 shows an example where the node for key part "A" would be omitted. To still be able to compare the full key Leis et al. identify three different approaches:

- Pessimistic: Each node can store a variable length part of the key in its header. Upon compression, the compressed key part is stored in the parent. When traversing a node, the additional stored key part is compared first, and then, in case of a match, we continue with the next partial key.
- Optimistic: Instead of storing the omitted key part itself, the parent node only stores the number of omitted nodes. Thus, traversing a node just skips this number of partial keys. When reaching a value, we will again have to compare the entire key to ensure the optimistically skipped key part matches.
- Hybrid: While the pessimistic approach does not require rechecking keys, it needs to store a variable length key part in its header leading to further cache misses when skipping long key parts. This differs from the optimistic approach, where each node only stores a single number.

ART uses the hybrid approach and stores the key part pessimistically for up to 8 bytes. If more than eight nodes get compressed, it switches to the optimistic approach.

2.3 Storing Values

Values associated with keys can be stored in different ways:

- Single-value leaves: The classical approach of storing a value in an extra leaf type node.
- Multi-value leaves: Multiple leaf node types based on the inner node types (Section 2.1) but storing values instead of child pointers.
- Combined pointer/value slots: When values fit the size of a pointer storing the value at the place instead of a child pointer can be done via pointer tagging or using an additional bit per pointer.

Single-value leaves increase the tree height but can differentiate keys of different lengths as opposed to multi-value leaves. Combined pointer/value slots is generally the best approach. On typical 64 bit architectures, the lowest bit of an alleged pointer can be set to indicate an actual stored value of up to 63 bits length as pointers are 8 byte aligned.

2.4 Algorithms & Space Consumption

The original ART paper [8] also shows and explains pseudo code for the search, insert, bulk loading and delete which is ommitted here for brevity.

Similarly, Leis et al. also presented a proof for the worst-case space consumption per key, which is bounded by 52 bits even for arbitrarily long keys. As we will see in Section 4.1, this space consumption per key can be in practice as low as 8.1 bits for dense keys.

3 CONSTRUCTING BINARY-COMPARABLE KEYS

To preserve order on keys in a radix tree and therefore allow range-based queries, partial keys have to be sorted. To allow for different spans and variable length keys, it is required that the bitwise representation of a key is ordered in the same way as the key in its own context. This is the case for some key types like ASCII encoded strings as the binary representation of 'A' is smaller than the one of 'B'. However, for general types, this may not be the case.

For this reason, Leis et al. proposed various key transformations for common types such that the binary representation of a type implies the same order as the type itself [8].

Having the same order means for two values a, b of the same type

$$a < b \iff bin(a) < bin(b) \qquad \text{(same for } > \text{ and } =\text{)}$$

where $bin(x)$ is the binary representation of x.

The transformation for the most common types are as follows:

- Unsigned Integers: The binary representation of unsigned integers is already ordered correctly. However, when operating on keys in memory, the endianness generally matters as for

little endian, the bytes have to be swapped first. This can also be done implicitly in the implementation.
- Signed Integers: For signed two-complement integers, the sign bit must first be flipped. Then it can be stored like an unsigned integer.
- Strings: The Unicode Collation Algorithm (UCA) already defines transformations for strings to binary-comparable keys.
- Compound Keys: Keys with multiple types can be transformed by transforming each type individually and concatenating the result.

Additionally the authors of ART also showed transformations for floats and nullable types [8].

4 EVALUATION

In this section, we evaluate ART first with our implementation and with microbenchmarks and compare the results to the same microbenchmarks used in the original ART paper before showing their results for performing the standard OLTP benchmark TPC-C.

Our ART implementation [4] uses combined value/pointer slots on 64 bit architecture with SIMD instructions for Node16 and does not include path compression, only storing a 2 byte header (node type and number of non-null children). Leaving out path compression does not result in any performance loss due to the keys of the microbenchmarks being only 32 bits long. At such short lengths, path compression generally introduces more overhead and memory consumption.

Our benchmarks were run on an Intel Core i5-8400 CPU with around 3.7 GHz reported clock rate during benchmarks. It has 6 cores, 6 threads, 8 MB shared, last-level cache, and 16 GB dual-channel DDR4-2666 RAM. As operation system, we used Windows 10.0.19044 and the MSVC v143 compiler with flags /O2 and /Ob3.

4.1 Mico-Benchmarks

The micro-benchmarks uses 56 thousand and 16 million 32 bit unsigned integers as keys without another associated value. We compare ARTs performance to a Trie implementation storing child pointers or the last partial key in a std::map (M-Trie), a Sorted List (SL), a chained hash-table (HT) using the std::unordered_set implementation, and a red-black tree (RB-Tree) using the std::set implementation. The results can be seen in Figure 6.

We split our datasets into dense and sparse keys where sparse are uniformly random 32 bit keys and dense keys only range from 0 to $n - 1$ where n is the number of keys inserted (16 million).

5 RELATED WORK

6 CONCLUSION AND FUTURE WORK

REFERENCES

[1] Doug Baskins. 2004. *Judy arrays.* Retrieved July 23, 2022 from http://judy.sourceforge.net/

[2] Matthias Boehm, Benjamin Schlegel, Peter Benjamin Volk, Ulrike Fischer, Dirk Habich, and Wolfgang Lehner. 2011. Efficient in-memory indexing with generalized prefix trees. In *Datenbanksysteme für Business, Technologie und Web (BTW)*, Theo Härder, Wolfgang Lehner, Bernhard Mitschang, Harald Schöning, and Holger Schwarz (Eds.). Gesellschaft für Informatik e.V., Bonn, 227–246.

[3] Jonathan Corbet. 2006. *Trees I: Radix trees.* Retrieved July 23, 2022 from https://lwn.net/Articles/175432/

[4] Jonas Fritsch. 2022. *Adaptive Radix Tree C++ Implementation.* Retrieved July 23, 2022 from https://github.com/atalantus/The-Adaptive-Radix-Tree/tree/main/Implementation

[5] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP through the Looking Glass, and What We Found There. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 981–992. https://doi.org/10.1145/1376616.1376713

[6] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 339–350. https://doi.org/10.1145/1807167.1807206

[7] Tobin J Lehman and Michael J Carey. 1985. *A study of index structures for main memory database management systems.* Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[8] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 38–49. https://doi.org/10.1109/ICDE.2013.6544812

[9] Donald R. Morrison. 1968. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM* 15, 4 (oct 1968), 514–534. https://doi.org/10.1145/321479.321481

[10] Jun Rao and Kenneth A Ross. 1998. Cache conscious indexing for decision-support in main memory. (1998). https://doi.org/10.7916/D8T441ZB

[11] Jun Rao and Kenneth A. Ross. 2000. Making B+- Trees Cache Conscious in Main Memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data* (Dallas, Texas, USA) *(SIGMOD '00)*. Association for Computing Machinery, New York, NY, USA, 475–486. https://doi.org/10.1145/342009.335449

[12] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. 2009. K-Ary Search on Modern Processors. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware* (Providence, Rhode Island) *(DaMoN '09)*. Association for Computing Machinery, New York, NY, USA, 52–60. https://doi.org/10.1145/1565694.1565705

[13] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. 2015. In-Memory Big Data Management and Processing: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (2015), 1920–1948. https://doi.org/10.1109/TKDE.2015.2427795

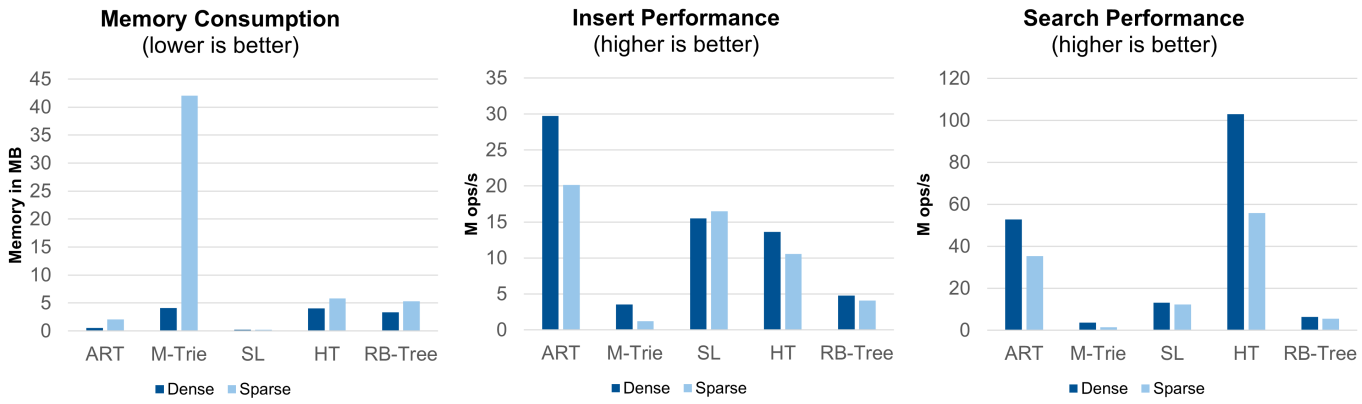

Fig. 5. Benchmark results for 56 thousand 32 bit unsigned integers.

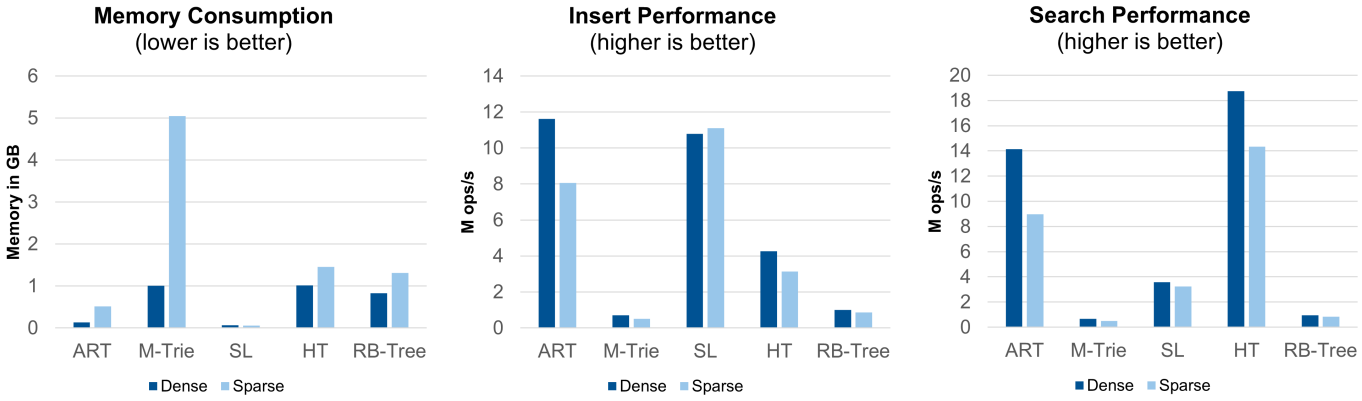

Fig. 6. Benchmark results for 16 million 32 bit unsigned integers.