The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases

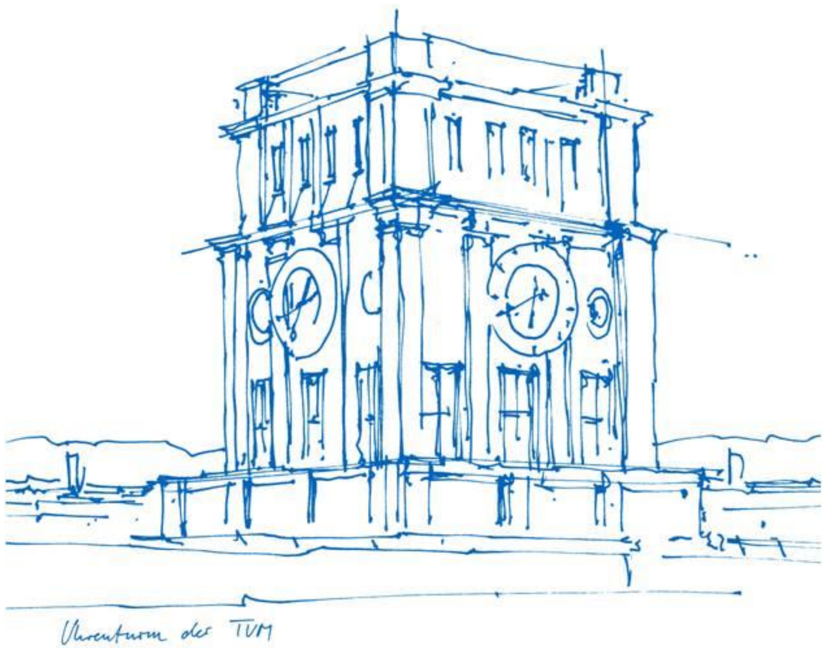
Jonas Fritsch

Technische Universität München

Department of Informatics

Chair for database systems

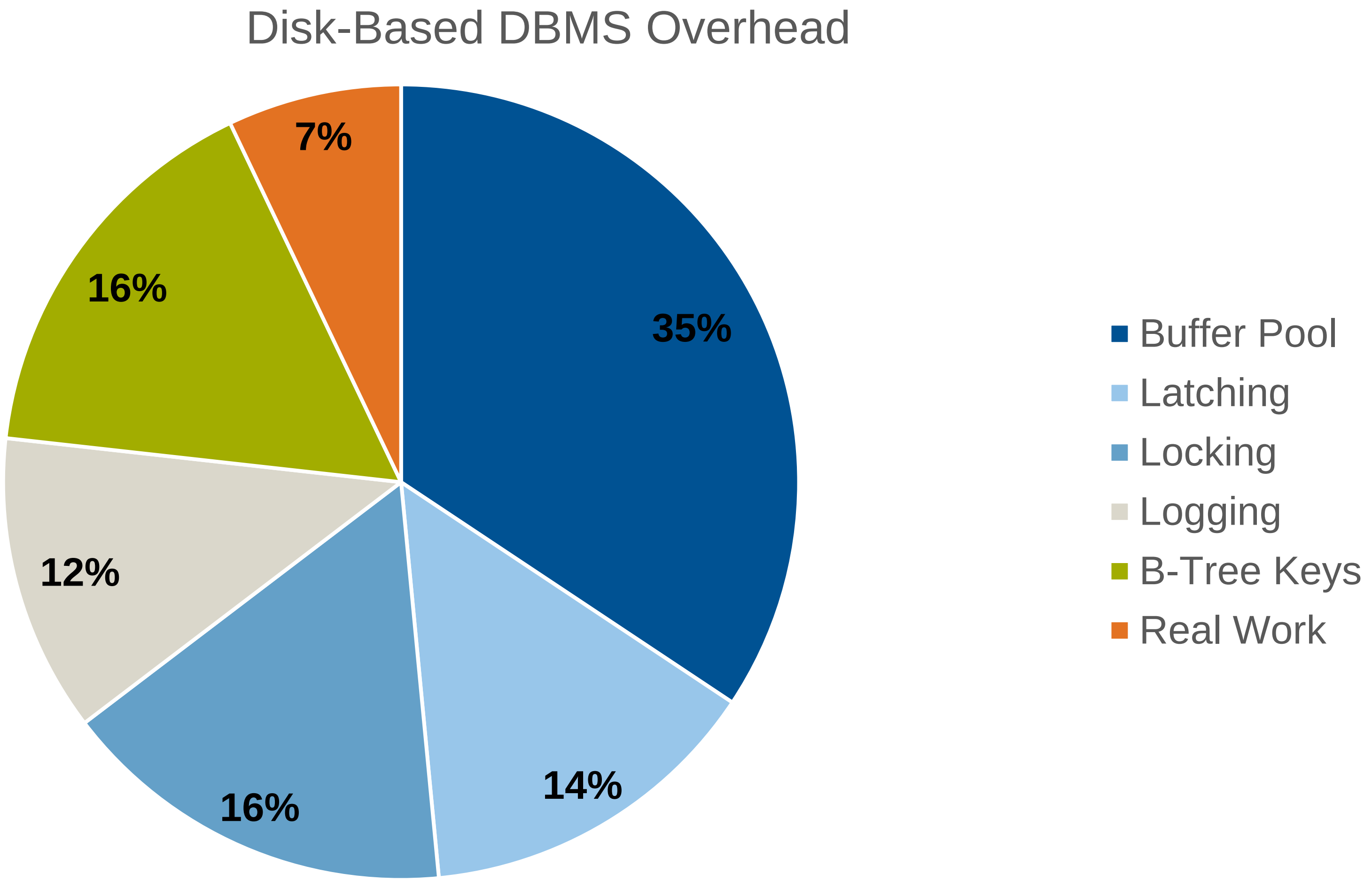
Munich, 11. July 2022



Overview

- Disk-Based vs. Main-Memory DBMS
- Index-Structures in DBMS
- From Tries to ART
- Key Transformations for Bitwise Comparisons
- Benchmarks
- Summary & Conclusion

Disk-Based vs Main-Memory DBMS



Source: OLTP through the looking glass, and what we found there. SIGMOD, 2008

Index-Structures in DBMS

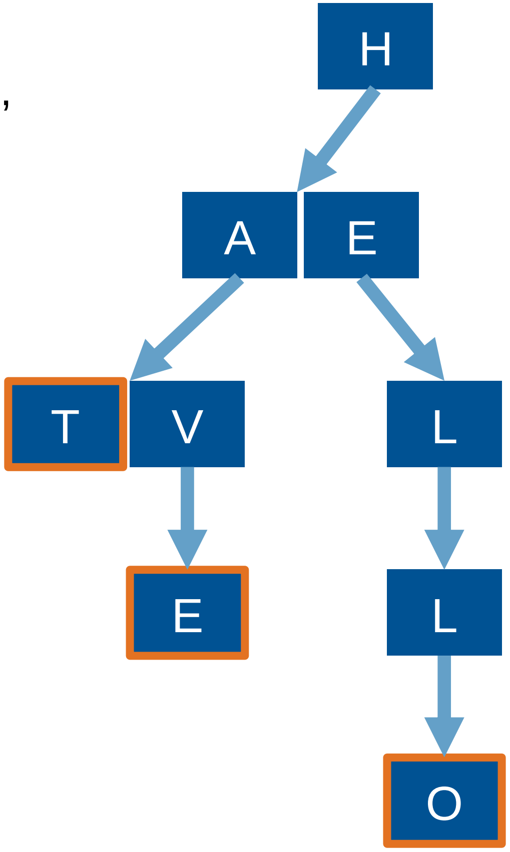
Data structures used to quickly find data in table via a key

2 Index-Types:

- Order Preserving Indexes (Maintains keys in sorted order)
- Hashing Indexes (Associative array mapping hash of key to data record. Only supports equality predicates!)

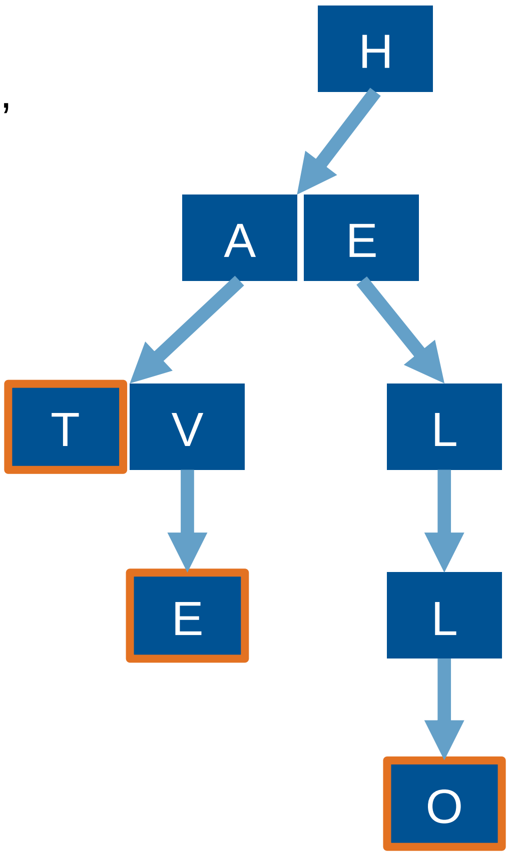
Tries

Keys:
HELLO,
HAT,
HAVE



Tries – Properties

Keys:
HELLO,
HAT,
HAVE

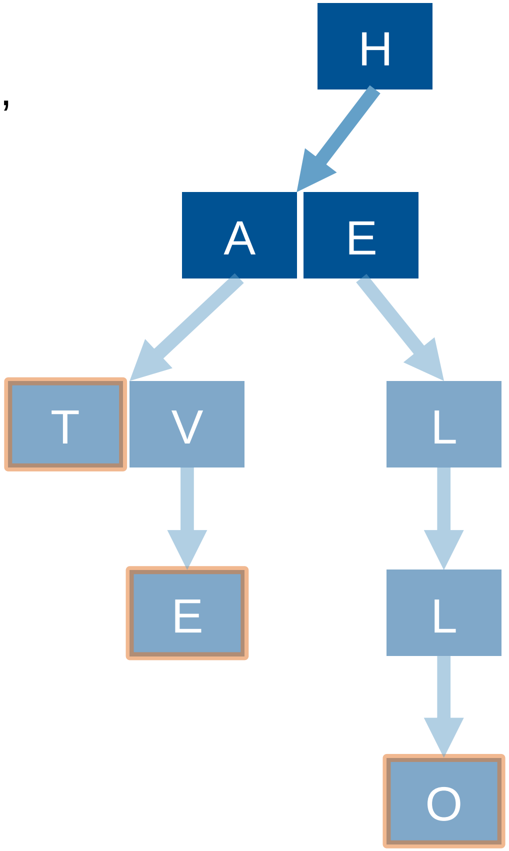


Properties:

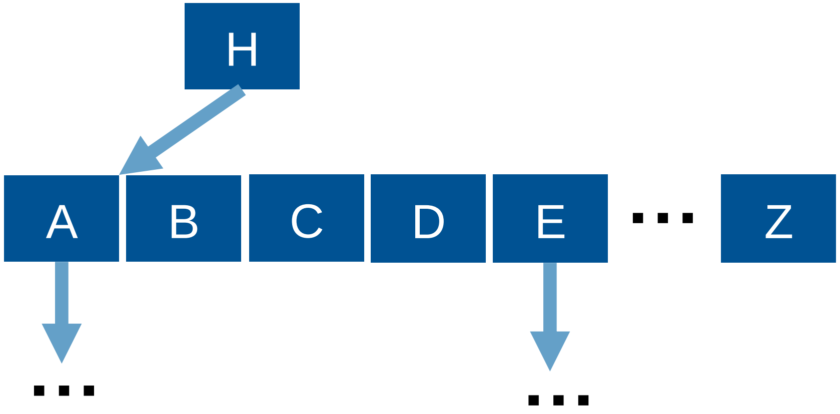
- Height/complexity depends on key length *k* instead of number of elements
- Require no rebalancing
- All insertion orders results in same tree
- Keys stored in lexicographic order
- Keys are stored implicitly along paths

Tries – Implementation

Keys:
HELLO,
HAT,
HAVE

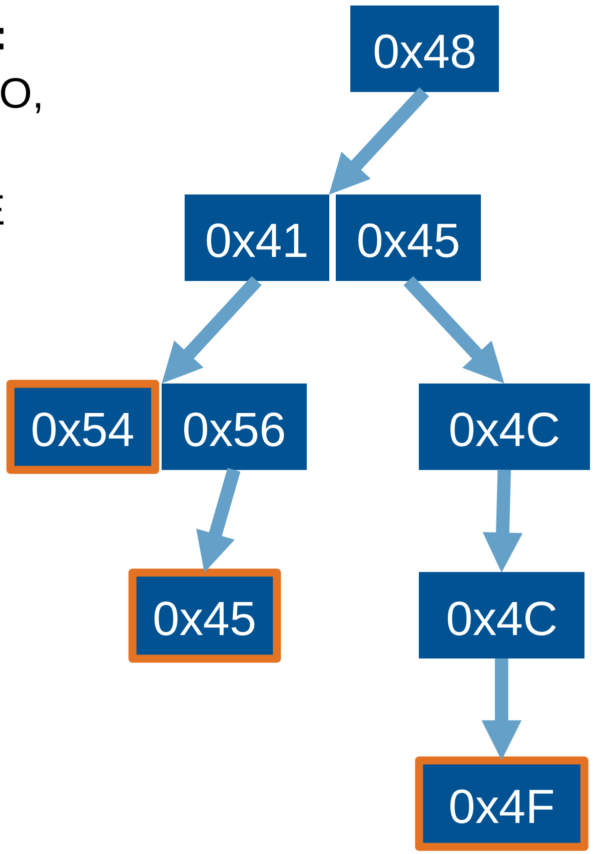



```
class Node {
    bool is_leaf;
    // ‘A‘ to ‘Z‘
    Node* children[26];
}
```



Tries – Key Span, Fanout and Height

Keys:
HELLO,
HAT,
HAVE



Key Span *s*:
Number of bits each partial key represents (e.g. char is 8 bit span)

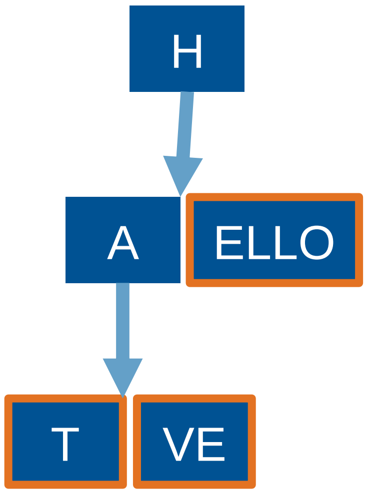
Fanout:
Number of children for a node

Determined by specific implementation but in general is 2^*s*

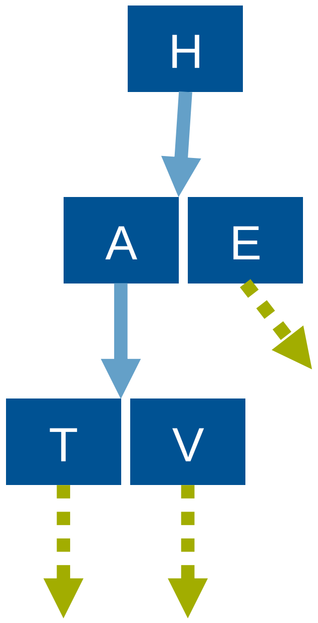
Height:
Max Height for *k* bit keys: ceil(*k*/*s*)

Radix Trees

Keys:
HELLO,
HAT,
HAVE

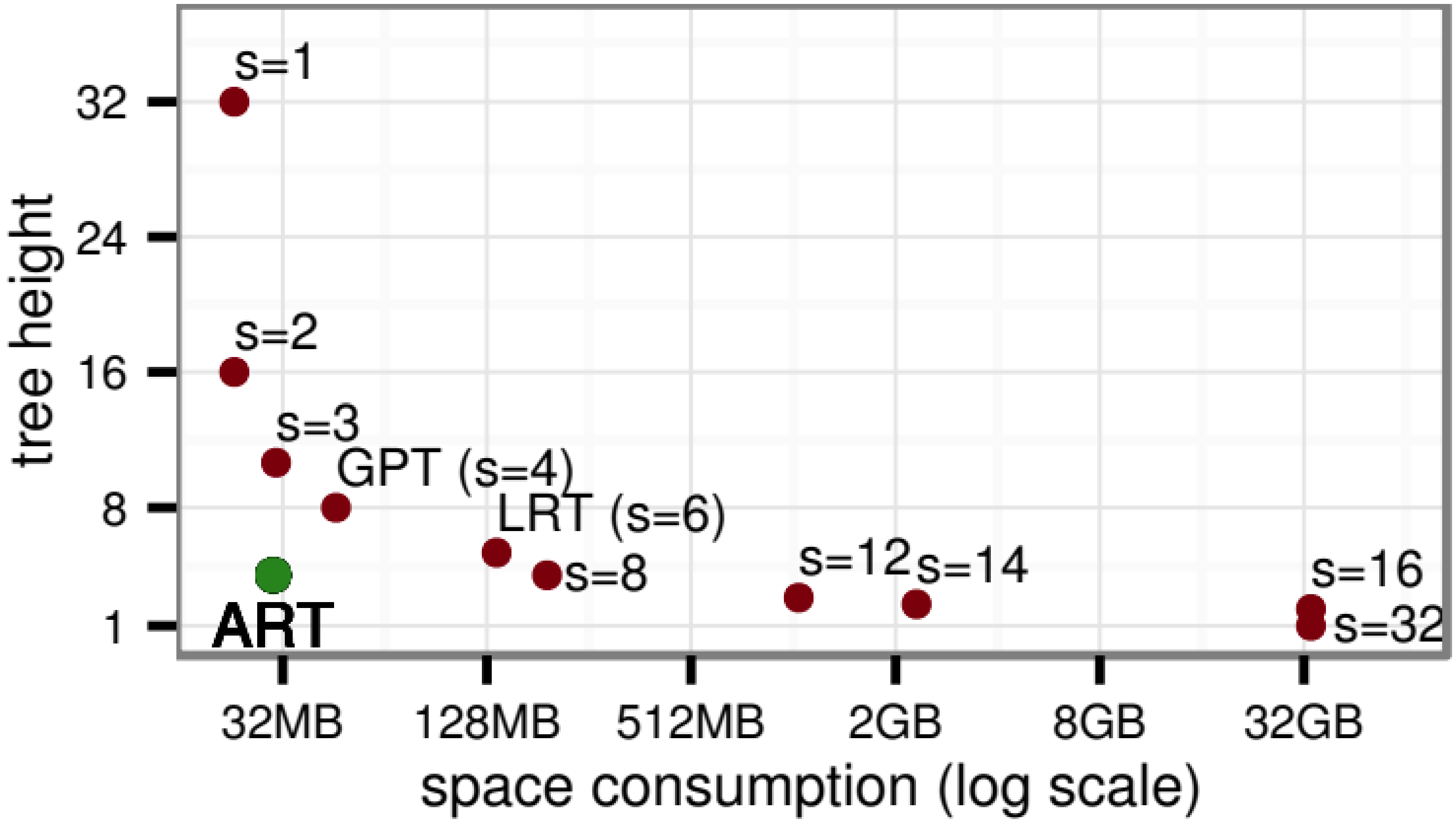


OR



Tuple
Pointer

Height vs. Space Tradeoff

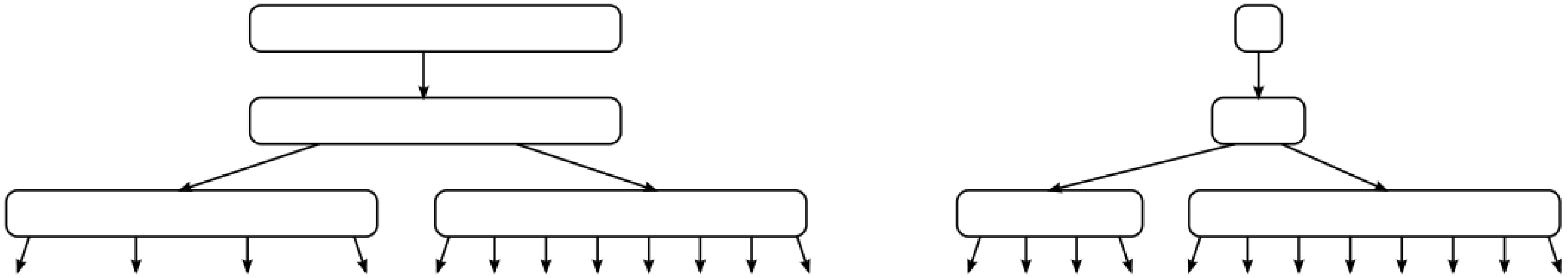


Source: [The adaptive radix tree: ARTful indexing for main-memory databases. IEEE, 2013]()

Adaptive Nodes

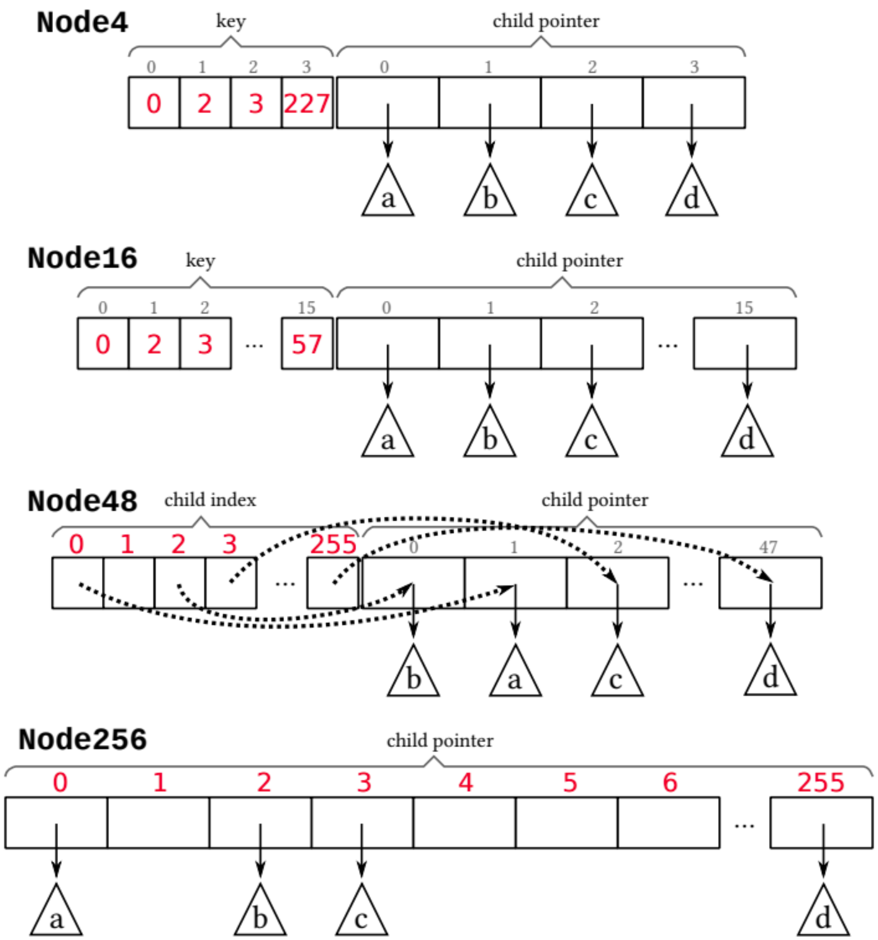
Main Idea:

Use different node types with different fanout based on number of non-null children.



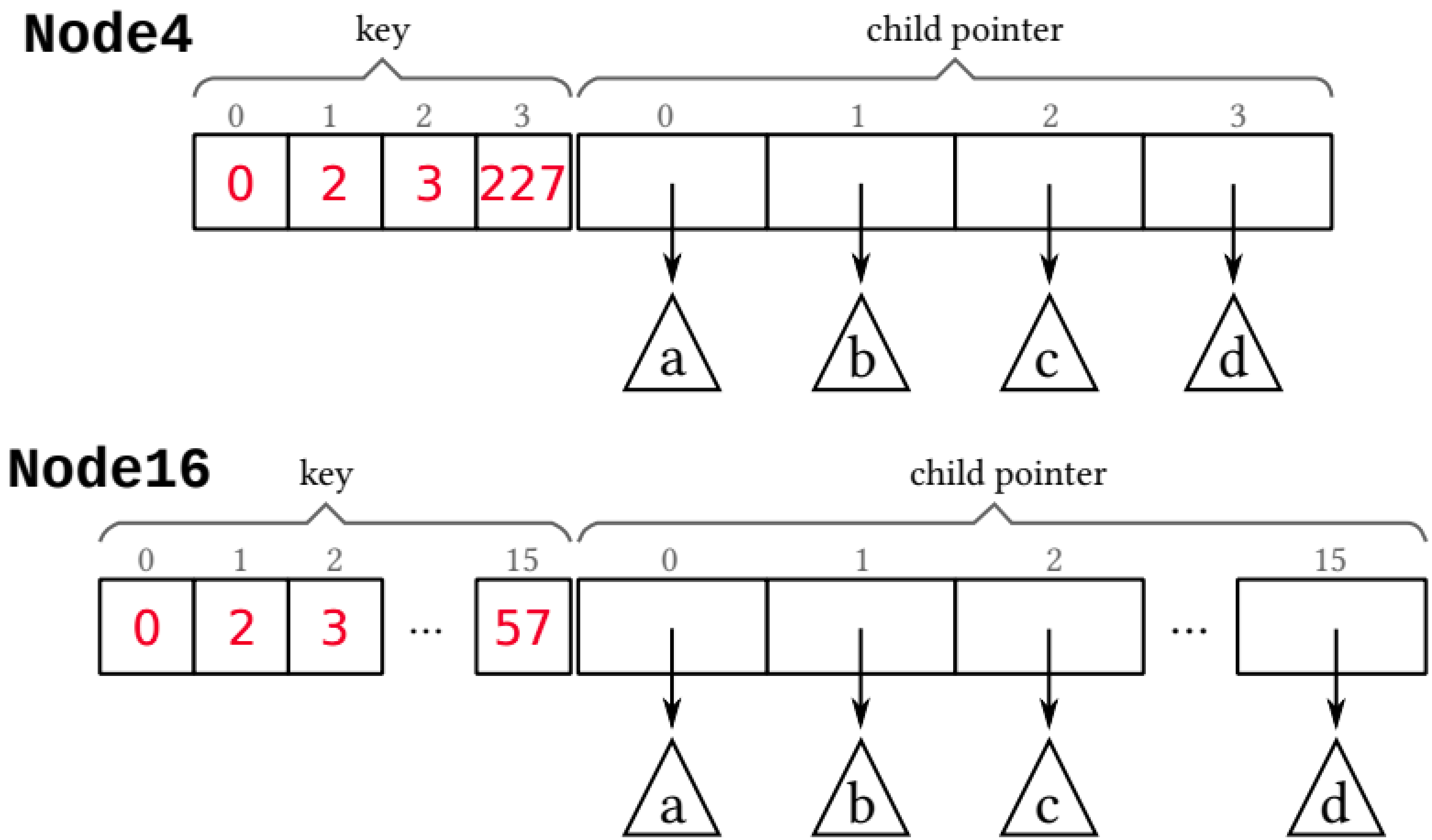
Source: [The adaptive radix tree: ARTful indexing for main-memory databases. IEEE, 2013]

ART – Adaptive Nodes



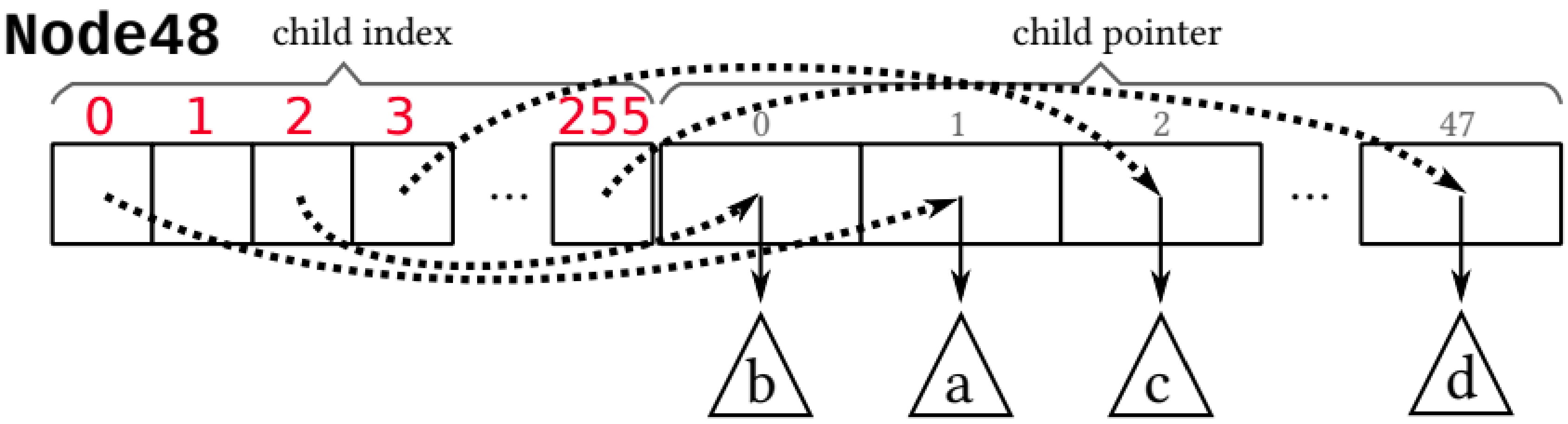
Source: [The adaptive radix tree: ARTful indexing for main-memory databases. IEEE, 2013]

ART – Adaptive Nodes



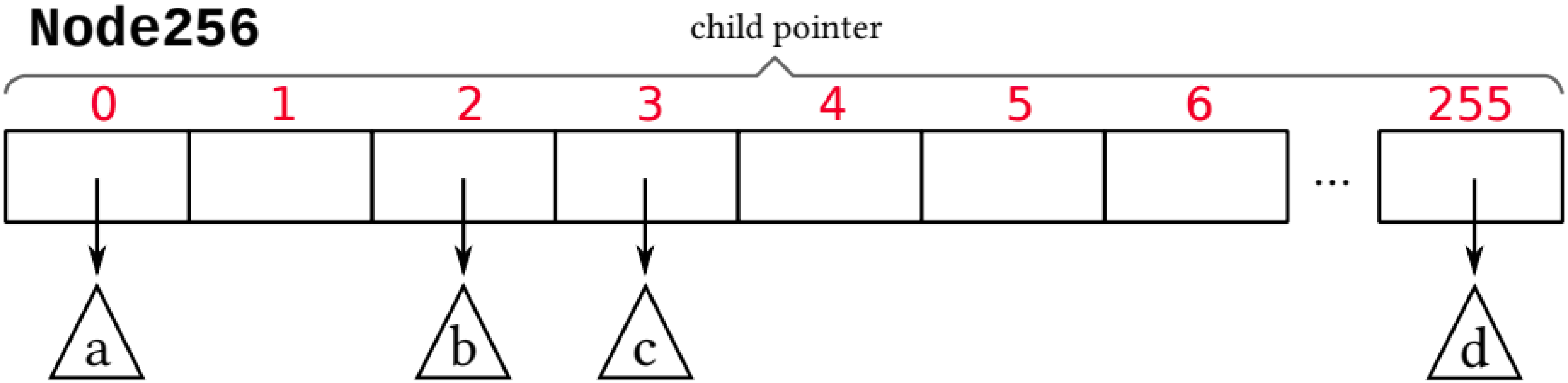
Source: The adaptive radix tree: ARTful indexing for main-memory databases. IEEE, 2013

ART – Adaptive Nodes



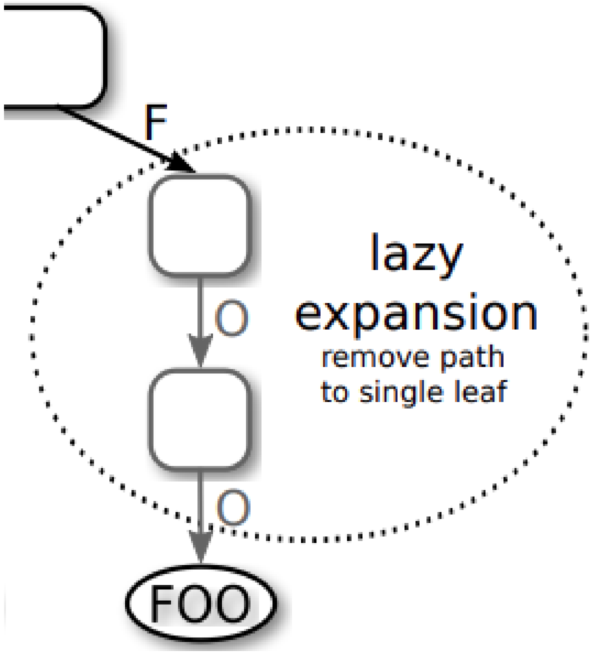
Source: The adaptive radix tree: ARTful indexing for main-memory databases. IEEE, 2013

ART – Adaptive Nodes



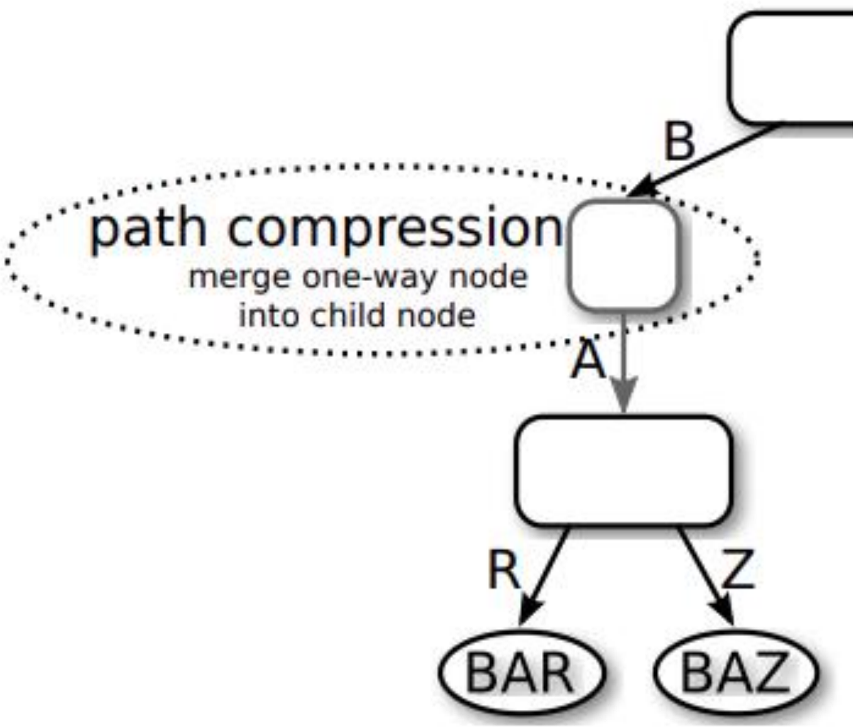
Source: [The adaptive radix tree: ARTful indexing for main-memory databases. IEEE, 2013]()

ART – Lazy Expansion



Source: [The adaptive radix tree: ARTful indexing for main-memory databases. IEEE, 2013]

ART – Path Compression



How to handle omitted partial key?

- **Pessimistic:**
  store omitted partial key at parent node
- **Optimistic:**
  only store count of omitted nodes and skip over equivalent key part (compare full key when reaching leaf)

Hybrid approach in ART:
Use pessimistic for size of up to 8 bytes for partial key. Then use optimistic.

Source: [The adaptive radix tree: ARTful indexing for main-memory databases. IEEE, 2013]

Key Transformations for Bitwise Comparisons

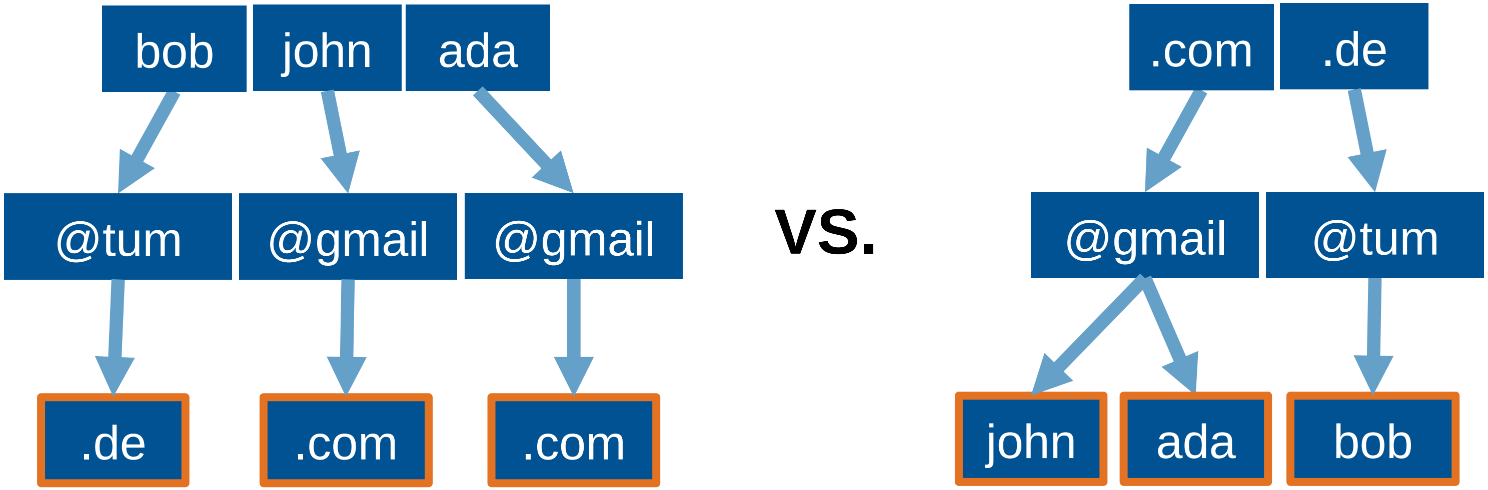
How to translate attribute types to make them binary comparible?

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

- **Unsigned Integers:** Flip byte order for little endian machines
- **Signed Integers:** Flip sign bit (so negative values come before positive ones), store as unsigned integer
- **Floats:** Classify into group (neg vs. Pos, normalized vs. denomarlized, NaN, ∞, 0), store as unsigned integer
- **Strings:** Library functions for Unicode Strings
- **Null:** Identify special value
- **Compound Type:** Transform each attribute separately, concatenate results

Store Attributes in Context Sensitive Way

Example: E-Mails from back to front for best compression



ART – Implementation Specifics

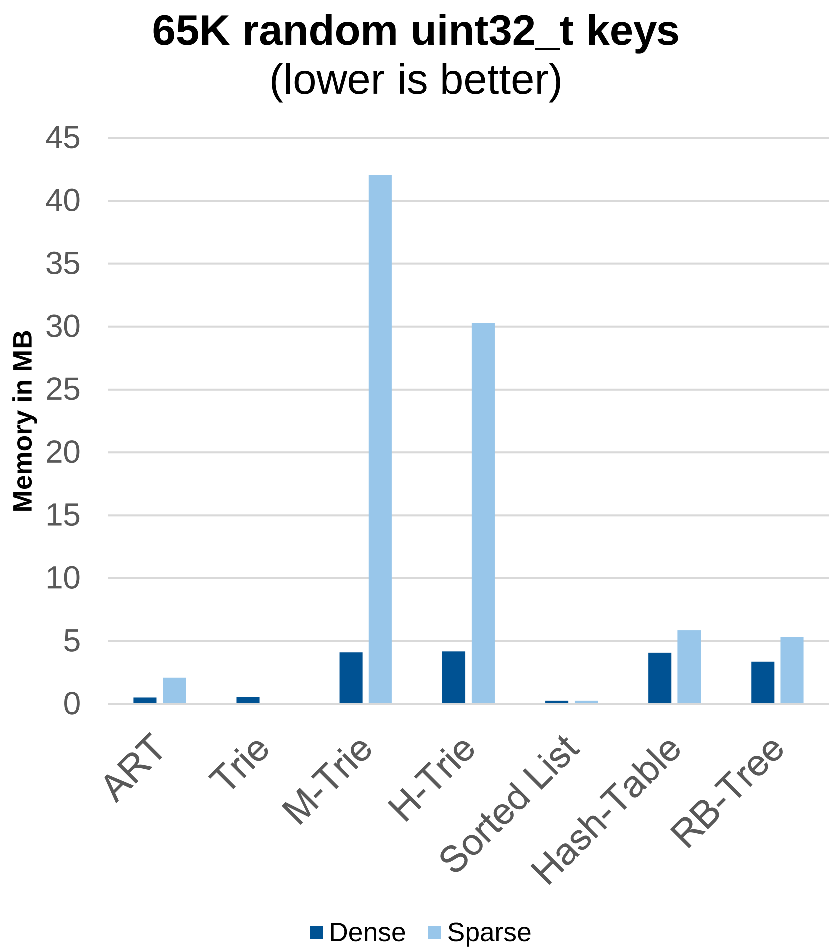
ART implementation for storing 32 bit keys (no path compression)

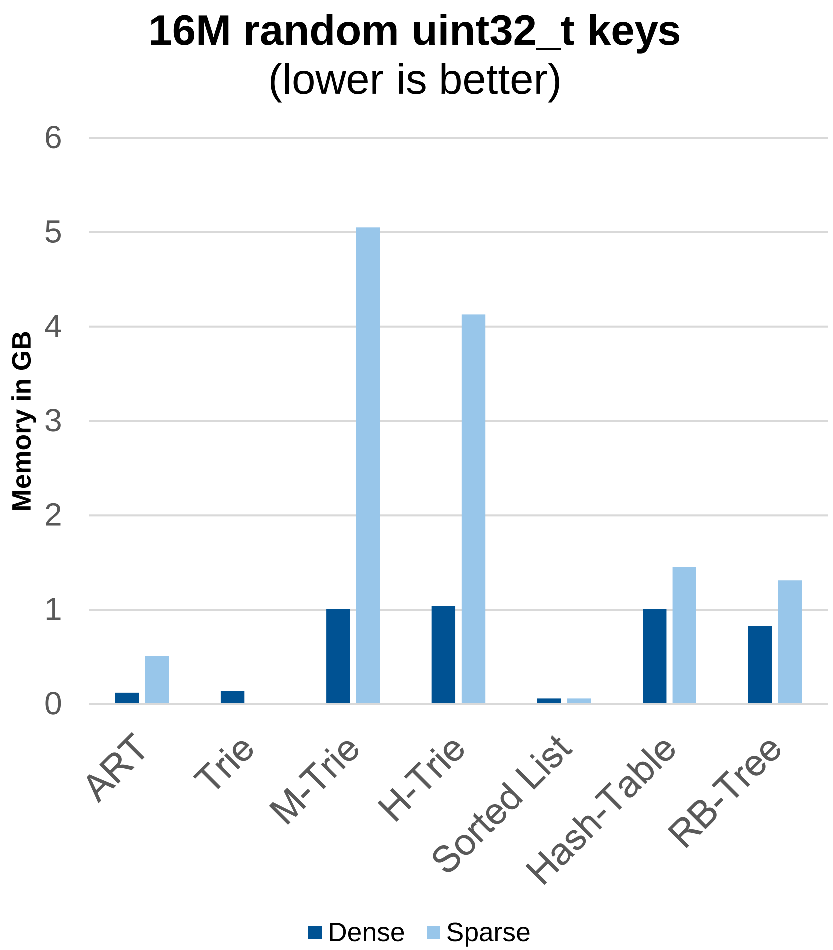
Important Points:

- No recursion! (self explanatory)
- Careful with polymorphie! (C++ utilizes vtables for dynamic dispatch → performance + memory overhead)
- Memory management! (malloc vs. TCMalloc vs. jemalloc vs. custom memory arenas/pools)
- Node16 search SIMD comparison: used x86-64 SSE2
- Combined value/pointer slots: Pointer Tagging using 3 low bits on 64 bit architecture (pointers are 8-byte aligned)

Source: https://github.com/atalantus/The-Adaptive-Radix-Tree/tree/main/Implementation

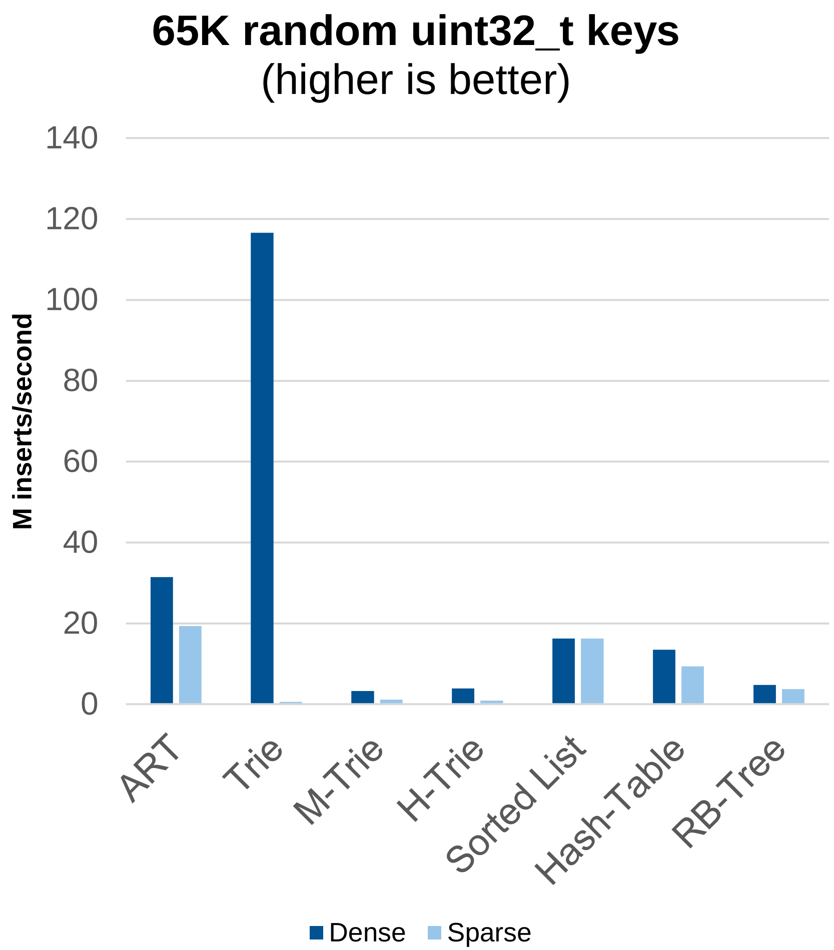
Memory Benchmark

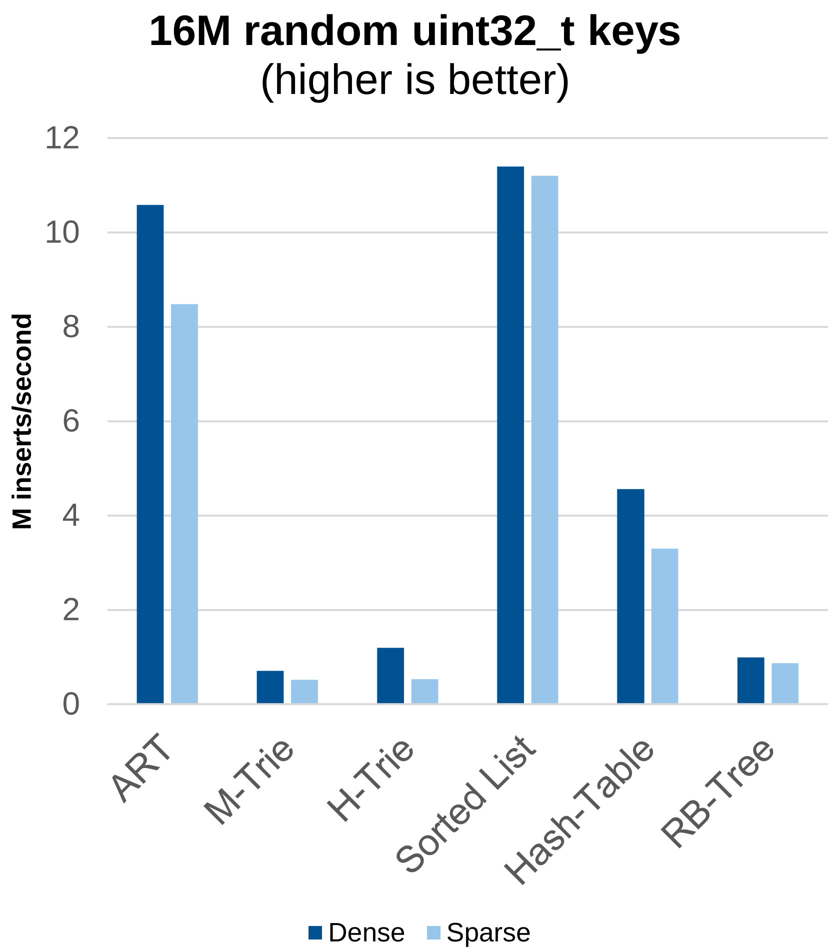




Source: https://github.com/atalantus/The-Adaptive-Radix-Tree/tree/main/Implementation

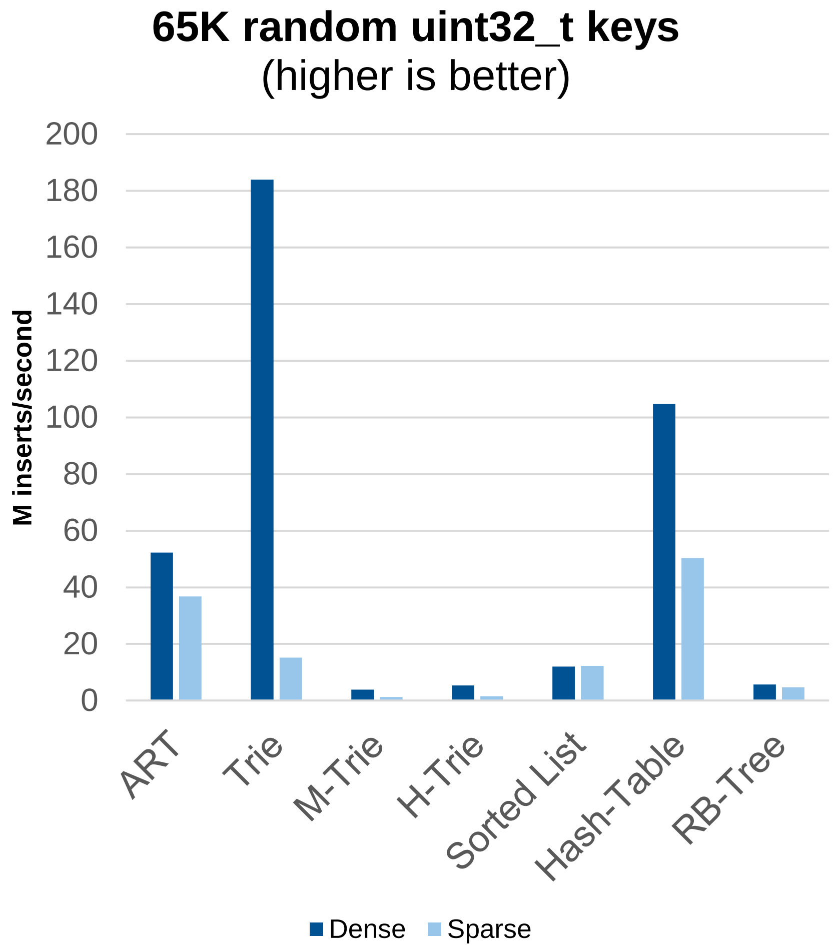
Performance Benchmark (insert)

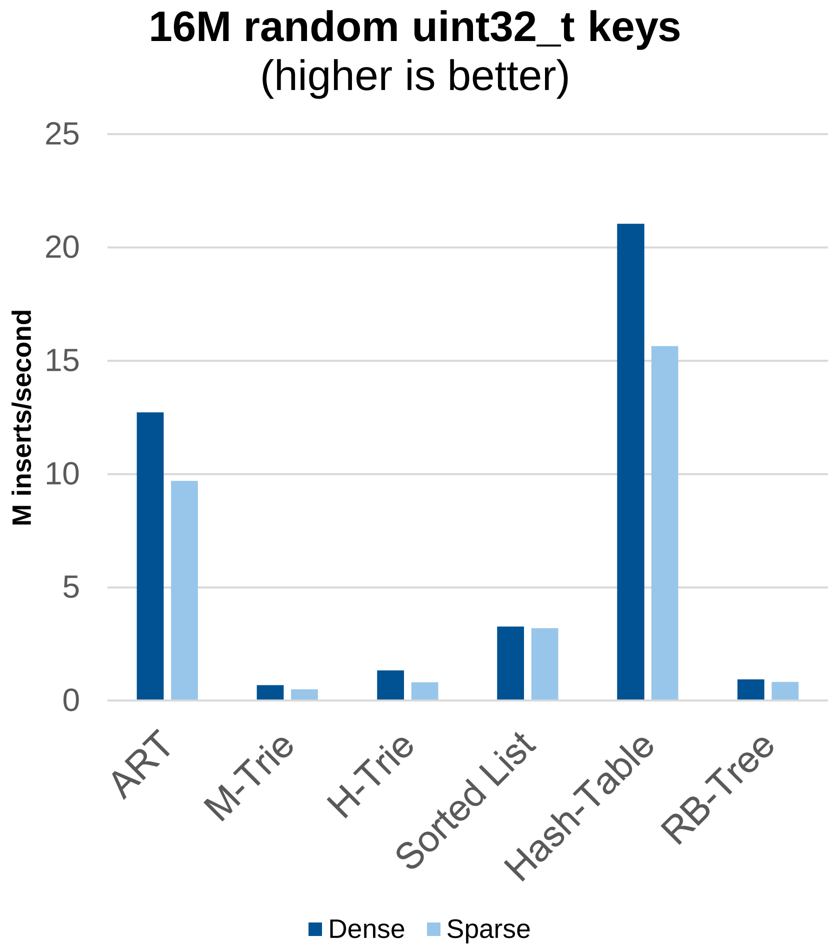




Source: https://github.com/atalantus/The-Adaptive-Radix-Tree/tree/main/Implementation

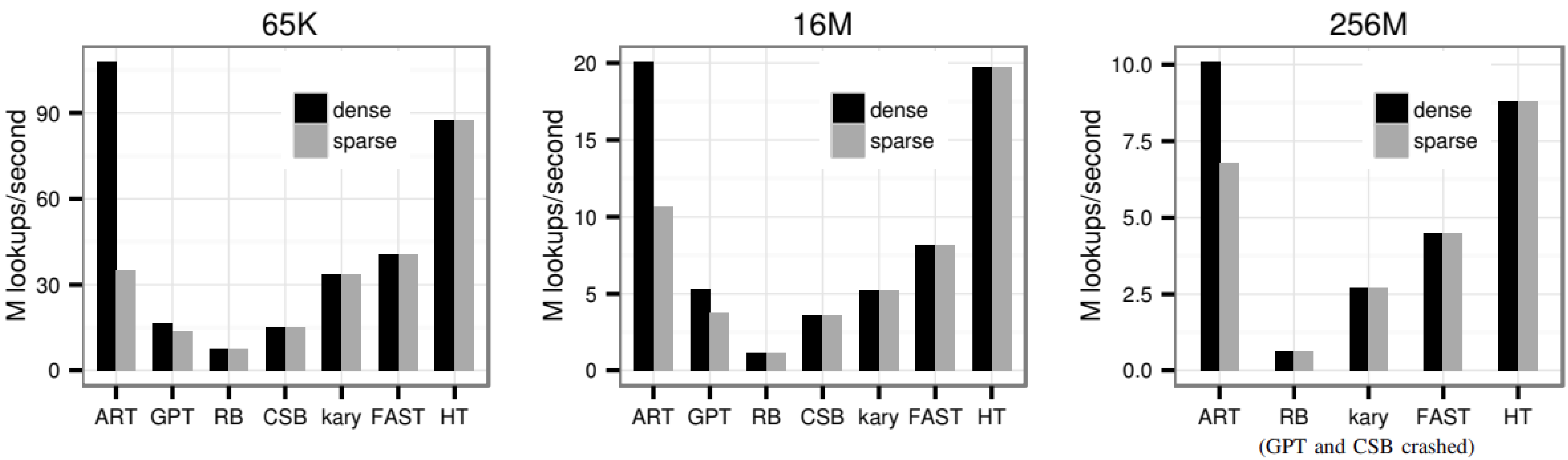
Performance Benchmark (search)





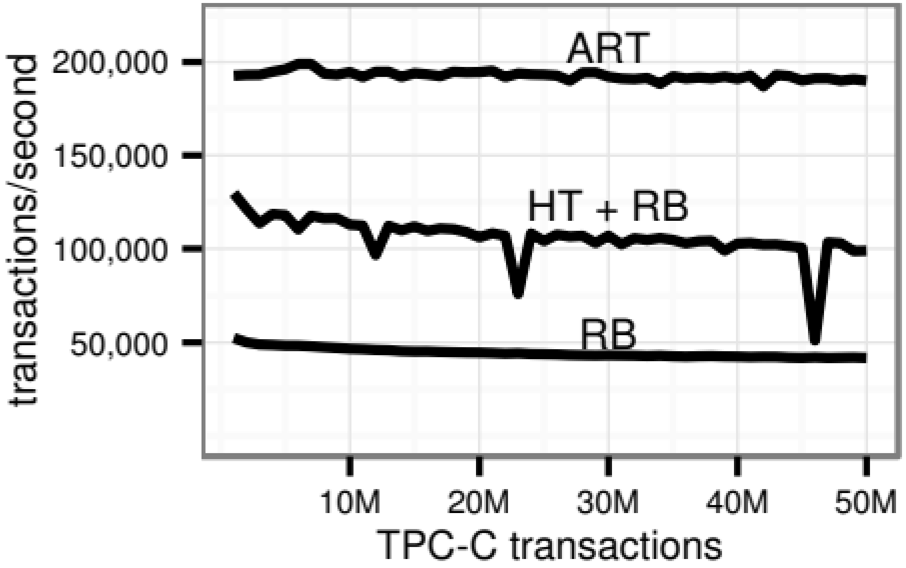
Source: https://github.com/atalantus/The-Adaptive-Radix-Tree/tree/main/Implementation

Performance Benchmark (search)



Source: [The adaptive radix tree: ARTful indexing for main-memory databases. IEEE, 2013]

TPC-C Benchmark



Source: The adaptive radix tree: ARTful indexing for main-memory databases. IEEE, 2013

Other Benchmarks: Memory Usage

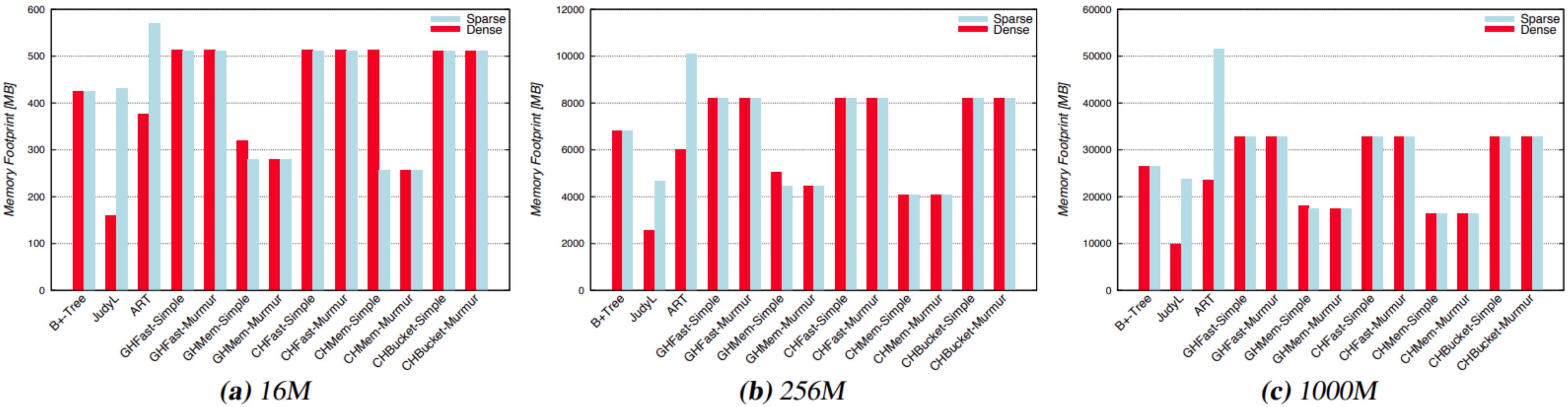


Figure 9: *Memory footprint in MB (covering). Lower is better.*

Source: [A comparison of adaptive radix trees and hash tables. IEEE 2015]

Other Benchmarks: Insert

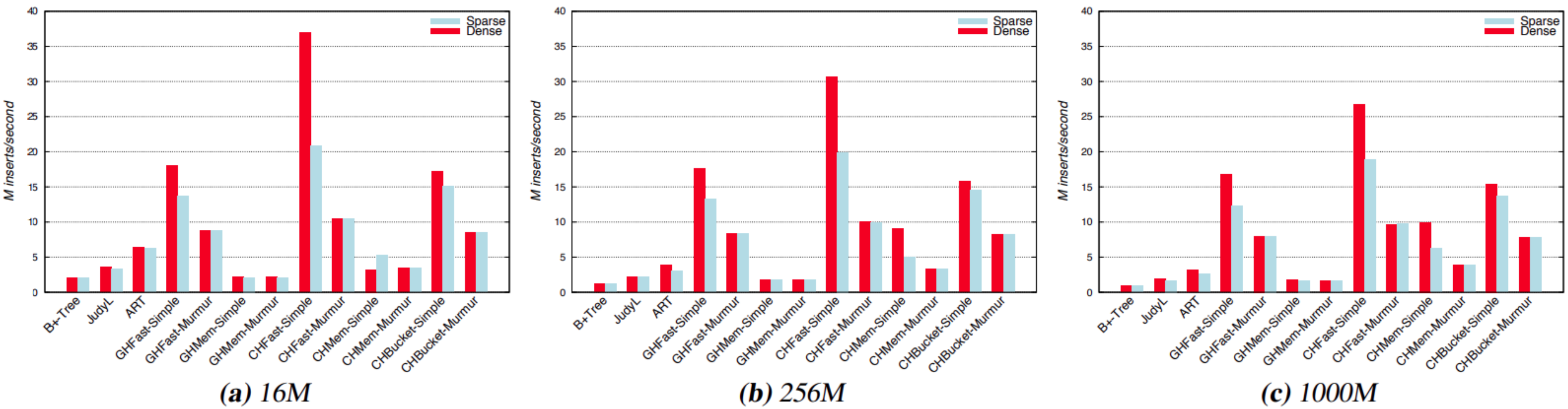


Figure 7: *Insertion throughput (covering). Higher is better.*

Source: [A comparison of adaptive radix trees and hash tables. IEEE 2015]

Other Benchmarks: Search

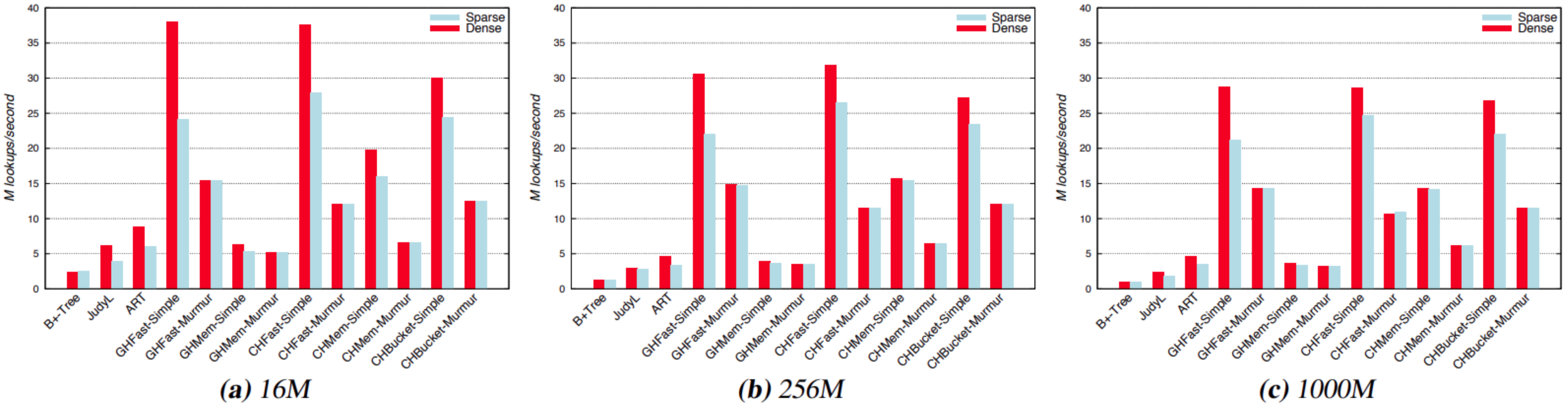
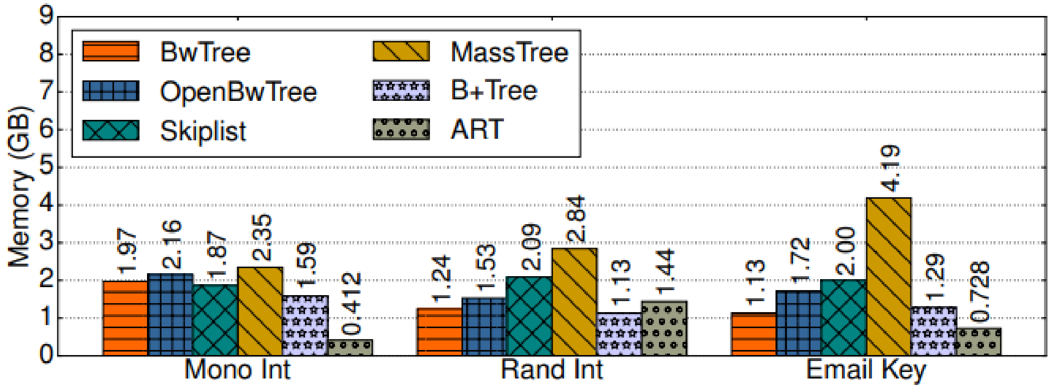


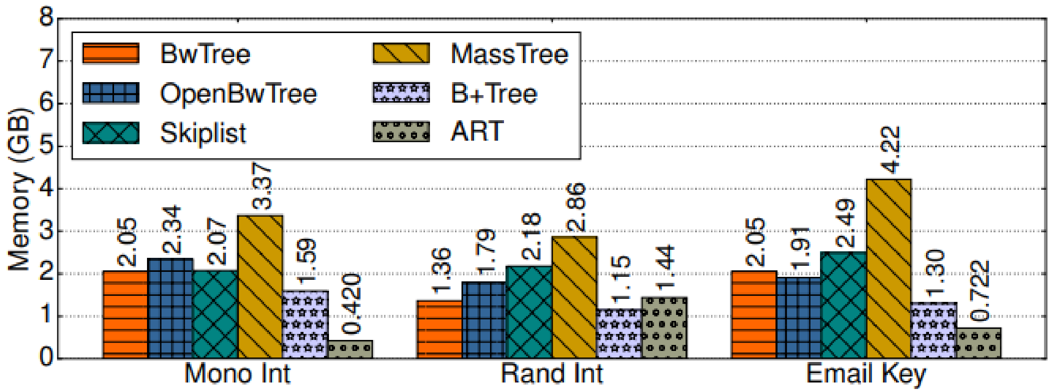
Figure 8: *Lookup throughput (covering). Higher is better.*

Source: [A comparison of adaptive radix trees and hash tables. IEEE 2015]

Other Benchmarks: Memory Usage



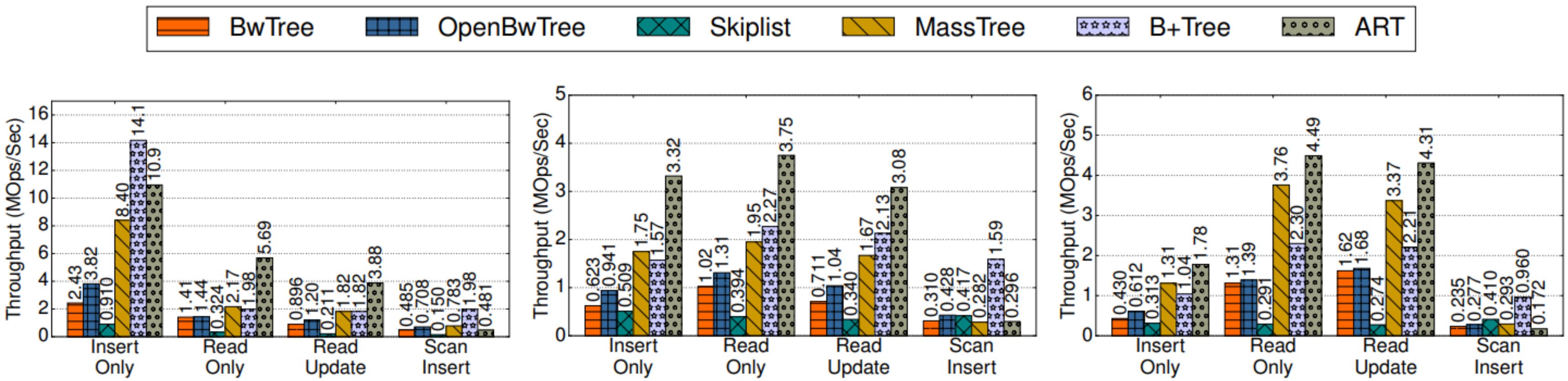
(a) Single-Threaded – Read/Update



(b) Multi-Threaded – Read/Update

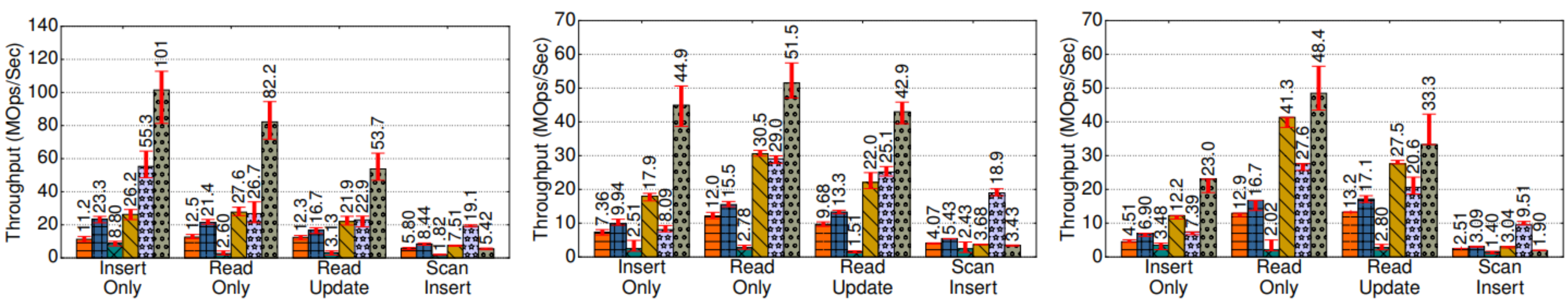
Source: Building a Bw-Tree Takes More Than Just Buzz Words. SIGMOD, 2018

Other Benchmarks: Performance



(a) Mono-Int Keys　**(b)** Rand-Int Keys　**(c)** Email Keys

Figure 13: In-Memory Index Comparison (Single-Threaded) – The worker thread is pinned to NUMA node 0.



(a) Mono-Int Keys　**(b)** Rand-Int Keys　**(c)** Email Keys

Figure 14: In-Memory Index Comparison (Multi-Threaded) – 20 worker threads. All worker threads are pinned to NUMA node 0.

Source: [Building a Bw-Tree Takes More Than Just Buzz Words. SIGMOD, 2018]

Summary & Conclusion

ART as (synchronized) order preserving index structure is

- space efficient
- outperforms other op-index structures in insert, search and update performance
- usable for typical data types
- used as default index in (at least) HyPer and DuckDB

	Range Support	Memory Efficiency	Performance	Researched
B+-Tree	+	−	−− (best for range)	+
ART	+	~	~	~
Judy-Array	+	+	−	−
Hash-Table	−	+	++	~

Summary & Conclusion

Further research being done:

The ART of practical synchronization (DaMoN, 2016)

- sychronize ART for multithreading using combination of OLP and ROWEX

HOT: A Height Optimized Trie Index for Main-Memory Database Systems (SIGMOD, 2018)

- ART performs worse on string (more random binary distribution) due to low fanout in lower tree levels
- Idea: Achieve consistently high fanout by adapting the span at each node

START – Self-Tuning Adaptive Radix Tree (IEEE, 2020)

- Learned indexes outperfoming ART in read-mostly benchmark due to hierarchical structure of nodes
- Idea: Optimize lower level nodes using cost model and optimizer

(Judy Arrays patent expired this year)

- even better cache efficiency, complex node transitions (~20k LoC ☹)
- Judy-Pointers

Benchmark Specs

Intel Core i5-8400 CPU @ 2.80GHz

• L1 Instruction Cache:	6 x 32 KB
• L1 Data Cache:	6 x 32 KB
• L2 Cache:	6 x 256 KB
• L3 Cache:	9 MB

16 GB DDR4 RAM

Windows 10 Pro (10.0.19044 Build 19044)

ART – Search

```
search (node, key, depth)
if node==NULL
  return NULL
if isLeaf(node)
  if leafMatches(node, key, depth)
    return node
  return NULL
if checkPrefix(node,key,depth)!=node.prefixLen
  return NULL
depth=depth+node.prefixLen
next=findChild(node, key[depth])
return search(next, key, depth+1)
```

Source: The adaptive radix tree: ARTful indexing for main-memory databases. IEEE, 2013

ART – Search

```
findChild (node, byte)
if node.type==Node4 // simple loop
  for (i=0; i<node.count; i=i+1)
    if node.key[i]==byte
      return node.child[i]
  return NULL
if node.type==Node16 // SSE comparison
  key=_mm_set1_epi8(byte)
  cmp=_mm_cmpeq_epi8(key, node.key)
  mask=(1<<node.count)-1
  bitfield=_mm_movemask_epi8(cmp)&mask
  if bitfield
    return node.child[ctz(bitfield)]
  else
    return NULL
if node.type==Node48 // two array lookups
  if node.childIndex[byte]!=EMPTY
    return node.child[node.childIndex[byte]]
  else
    return NULL
if node.type==Node256 // one array lookup
  return node.child[byte]
```

Source: The adaptive radix tree: ARTful indexing for main-memory databases. IEEE, 2013

ART – Insert

```
insert (node, key, leaf, depth)
if node==NULL // handle empty tree
  replace(node, leaf)
  return
if isLeaf(node) // expand node
  newNode=makeNode4()
  key2=loadKey(node)
  for (i=depth; key[i]==key2[i]; i=i+1)
    newNode.prefix[i-depth]=key[i]
  newNode.prefixLen=i-depth
  depth=depth+newNode.prefixLen
  addChild(newNode, key[depth], leaf)
  addChild(newNode, key2[depth], node)
  replace(node, newNode)
  return
p=checkPrefix(node, key, depth)
```

Source: [The adaptive radix tree: ARTful indexing for main-memory databases. IEEE, 2013]

ART – Insert

```
if p!=node.prefixLen // prefix mismatch
  newNode=makeNode4()
  addChild(newNode, key[depth+p], leaf)
  addChild(newNode, node.prefix[p], node)
  newNode.prefixLen=p
  memcpy(newNode.prefix, node.prefix, p)
  node.prefixLen=node.prefixLen-(p+1)
  memmove(node.prefix,node.prefix+p+1,node.prefixLen)
  replace(node, newNode)
  return
depth=depth+node.prefixLen
next=findChild(node, key[depth])
if next // recurse
  insert(next, key, leaf, depth+1)
else // add to inner node
  if isFull(node)
    grow(node)
  addChild(node, key[depth], leaf)
```

Source: [The adaptive radix tree: ARTful indexing for main-memory databases. IEEE, 2013]

ART – Space Consumption

TABLE I
SUMMARY OF THE NODE TYPES (16 BYTE HEADER, 64 BIT POINTERS).

Type	Children	Space (bytes)
`Node4`	2-4	$16 + 4 + 4 \cdot 8 = 52$
`Node16`	5-16	$16 + 16 + 16 \cdot 8 = 160$
`Node48`	17-48	$16 + 256 + 48 \cdot 8 = 656$
`Node256`	49-256	$16 + 256 \cdot 8 = 2064$

TABLE II
WORST-CASE SPACE CONSUMPTION PER KEY (IN BYTES) FOR DIFFERENT RADIX TREE VARIANTS WITH 64 BIT POINTERS.

	$k = 32$	$k \to \infty$
ART	43	52
GPT	256	∞
LRT	2048	∞
KISS	$>$4096	NA.

Source: [The adaptive radix tree: ARTful indexing for main-memory databases. IEEE, 2013]

Performance-Benchmark (search)

TABLE III
PERFORMANCE COUNTERS PER LOOKUP.

	65K			16M		
	ART (d./s.)	FAST	HT	ART (d./s.)	FAST	HT
Cycles	40/105	94	44	188/352	461	191
Instructions	85/127	75	26	88/99	110	26
Misp. Branches	0.0/0.85	0.0	0.26	0.0/0.84	0.0	0.25
L3 Hits	0.65/1.9	4.7	2.2	2.6/3.0	2.5	2.1
L3 Misses	0.0/0.0	0.0	0.0	1.2/2.6	2.4	2.4

Source: [The adaptive radix tree: ARTful indexing for main-memory databases. IEEE, 2013]