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

V. Leis, A. Kemper, T. Neumann, **2013 ICDE**

2024. 02. 21

Presentation by Nakyeong Kim, Suhwan Shin

nkkim@dankook.ac.kr, shshin@dankook.ac.kr

Contents

1. Motivation

Motivation

- As main memory capacity has increased, storing data in RAM has become less of a challenge
- The bottleneck in in-memory databases is index performance
- Traditional comparison-based indexes(e.g., B+Tree) are not well suited for modern hardware architecture (e.g., branch misprediction)
- Hash is also fast, but only supports non-ordered operations (e.g., point query)

2.1. Radix Tree

Prefix Tree / Trie

- Non-comparison-based search tree
  Inner node is an array of 2^s pointers (*s*: span)
  s is used as the index to determine next child node without any comparison

- No rebalancing operations

- Height depends on the length of keys, not on the number of elements O(log*k*)

- Keys are stored in lexicographical order

- Key equals the path to a leaf node
  Inner nodes map partial keys to other nodes (path compression)

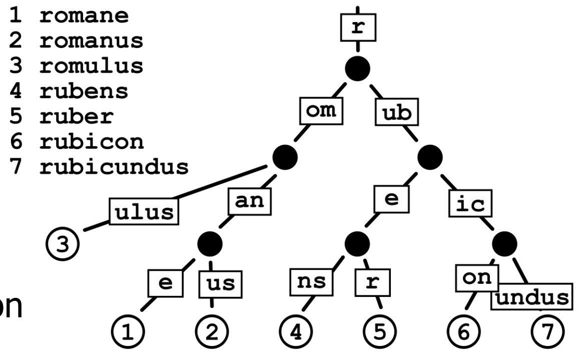


Fig. Radix Tree

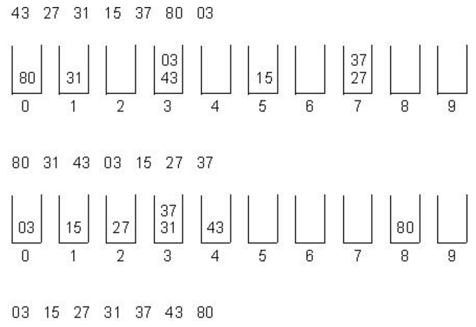


Fig. Radix Sort

2.2. Adaptive Radix Tree

Features

- ART was developed to provide an indexing structure that efficiently utilizes CPU caches
  - Supports fast insertions and deletions, and maintains sorted of data for additional operations

- ART achieves space and time efficiency by **adaptively choosing compact data structures**
  - Solving the problem of excessive space consumption that affects most radix trees

3.1. Adaptive Nodes

Adaptive Node

- There is a trade-off between performance (large span) and space consumption (null proportion) in radix tree
- ART uses different fanouts for each node to achieve both space and time efficiency
- Use a small number of node types (having different capacity) to reduce cost of resizing node

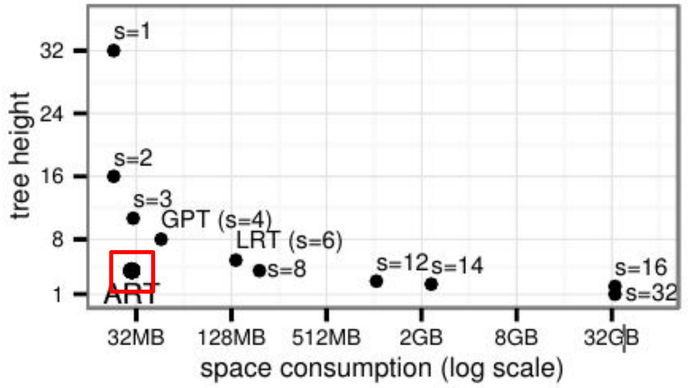


Fig. Space Consumption per Span

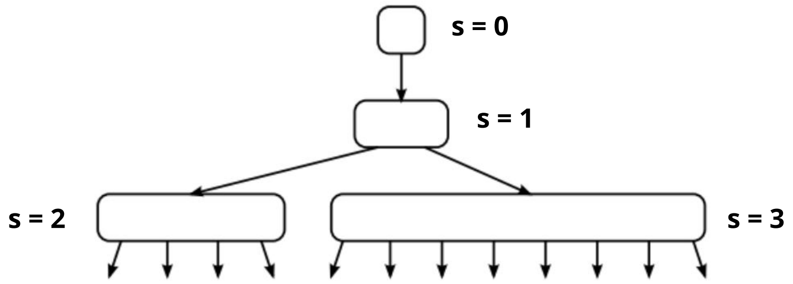


Fig. Adaptive Nodes of ART

3.2. Structure of Inner Nodes

Node Types

- **Node4**
  - Store up to 4 child pointers
  - 4 length array for keys
- **Node16**
  - Store 5~16 child pointers
  - 16 length array for keys
  - Use binary search or SIMD for lookup
- **Node48**
  - 256 length array for child pointers
  - Store 17~48 child pointers (secondary)
- **Node256**
  - Store 49~256 child pointers
  - 256 length array for keys

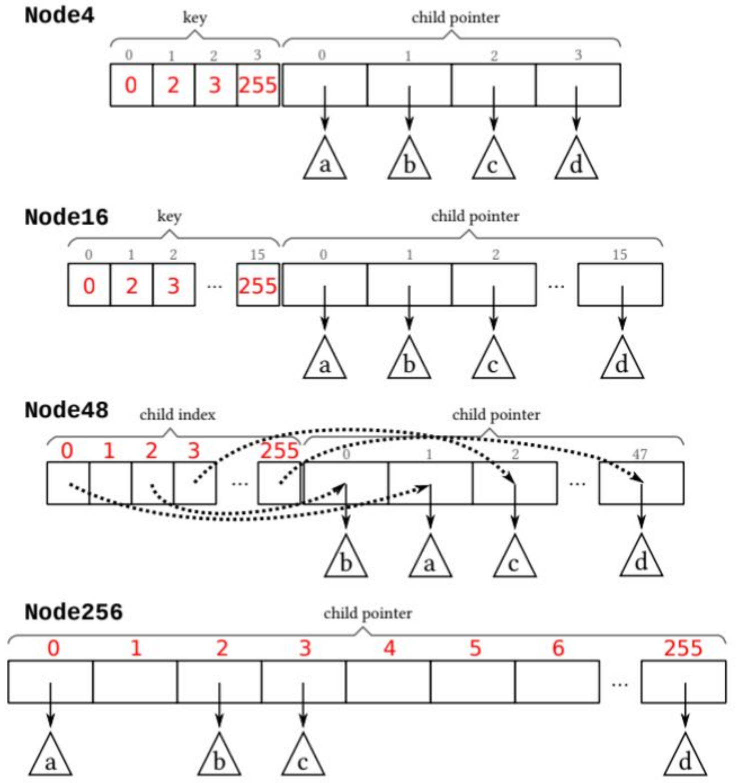


Fig. 5. Data structures for inner nodes. In each case the partial keys 0, 2, 3, and 255 are mapped to the subtrees a, b, c, and d, respectively.

4.1. Collapsing Inner Nodes

Reducing Space Consumption

- Path to leaf node is partial keys (Implicit prefix compression)

- Longer keys still consume a lot of space

- For long keys, two additional techniques are used
  It ensures that each inner node has at least two children
  - Lazy expansion
  - Path compression

- It decreases height of tree and reduce space consumption

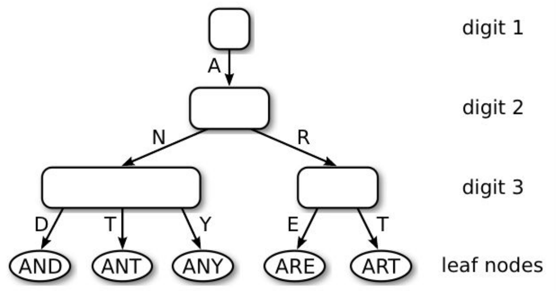
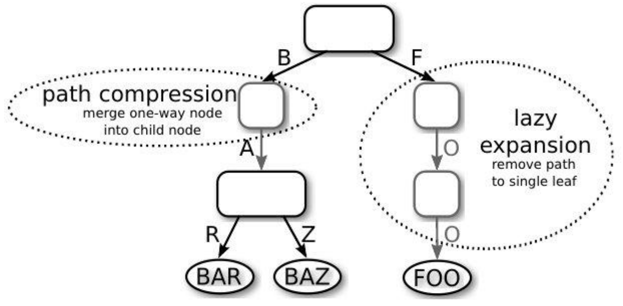


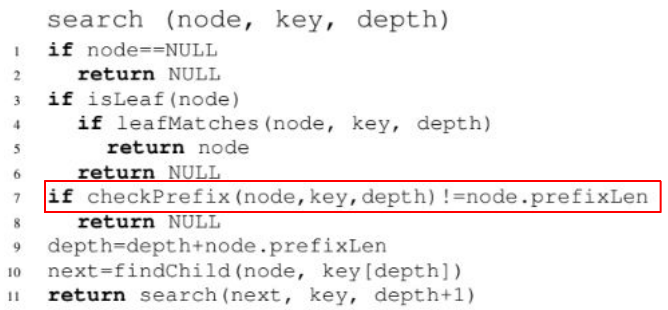
Fig. 1. Adaptively sized nodes in our radix tree.



B
F
path compression
merge one-way node into child node
A
lazy expansion
remove path to single leaf
O
O
R
Z
BAR
BAZ
FOO

Fig. 6. Illustration of lazy expansion and path compression.

4.1. Collapsing Inner Nodes



```
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)
```

Path Compression

- **Pessimistic**
  - Each inner node contains a variable length partial key vector (removed node's keys)
  - During lookup, check this vector before proceeding to the next child
  - Use more space and lead increased memory fragmentation as variable sized nodes

- **Optimistic**
  - Only the count of nodes(length of vector) are stored
  - During lookup, skip the number of bytes without comparison

    When reached to leaf node, its key must be compared to ensure "wrong turn" was taken
  - Beneficial for long string keys, but require additional check

Use **hybrid approach** by storing a vector at each node like pessimistic approach, but with a constant size for all
When this size is exceeded, switched to optimistic strategy

Dense: integers ranging from 1 to n
Sparse: many pointers of inner nodes are null

4.2. Space Consumption

By Distribution of Stored Keys

- Dense keys are the best space-efficient
- **Adaptive nodes** ensure that any key distribution is stored compactly
- 16 bytes for header (all nodes), 8 bytes for pointer
- Space consumption is 8.1 to 52 bytes (bounded)

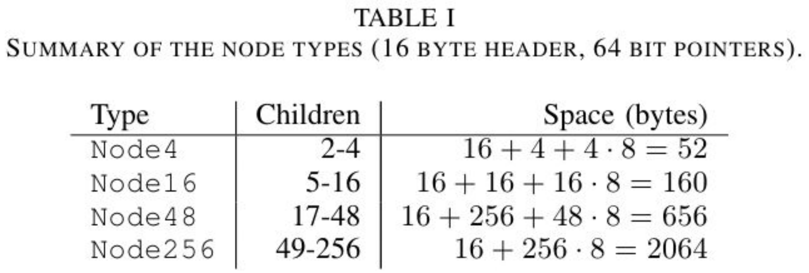


TABLE I
SUMMARY OF THE NODE TYPES (16 BYTE HEADER, 64 BIT POINTERS).

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	$16 + 256 \cdot 8 = 2064$

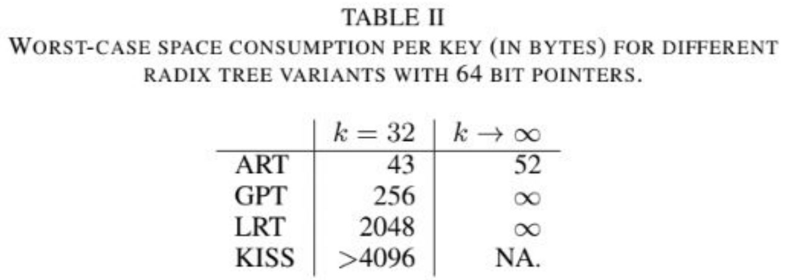


TABLE II
WORST-CASE SPACE CONSUMPTION PER KEY (IN BYTES) FOR DIFFERENT RADIX TREE VARIANTS WITH 64 BIT POINTERS.

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

5. Evaluation

32 Bit Integer Keys

- **Contestant**
  - Non-comparison-based tree: ART
  - Comparison-based trees: Cash Sensitive B -Tree(CSB), Red-Black Tree(RBT)
  - Read-Only trees: FAST, k-ary Search
  - Hashing: Chained Hash Table(HT)
  - Radix Trees: Generalized Prefix Tree(GPT)

- **Workload**
  - Standard OLTP

- **Environment**
  - Intel Core i7 3930K 6 cores(12 threads), 3.2 GHz(3.8 GHz turbo)
  - 12MB shared and last-level cache, 32GB quad-channel DDR3-1600 RAM
  - Linux 3.2 in 64 bit mode, GCC 4.6

Index Size: 65K, 16M, 256M
Key Pattern: Random, Existing

5.1. Search Performance

Single-Threaded

- ART performs very well when keys are dense and index sizes are large, while hash tables are stable and perform well in all cases (dense and sparse)

- Small size of indexes are much faster because of caching effects (determined by the number of instructions and branch mispredictions)

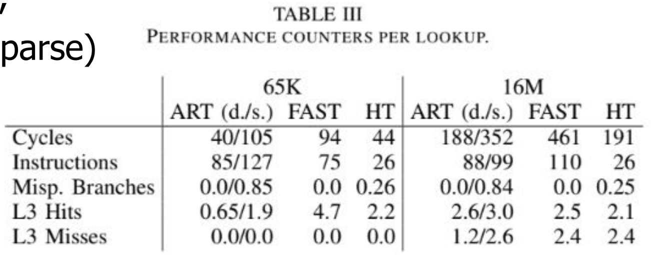


TABLE III
PERFORMANCE COUNTERS PER LOOKUP.

	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

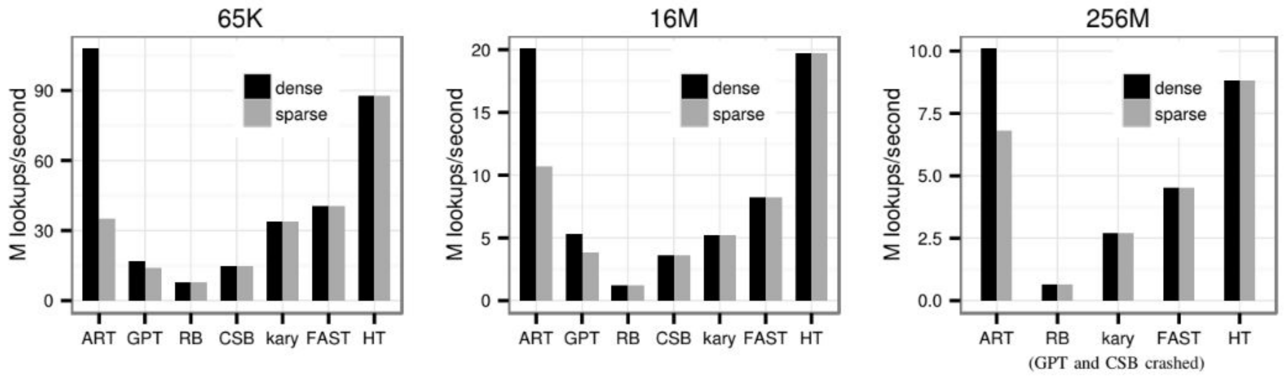


Fig. 10. Single-threaded lookup throughput in an index with 65K, 16M, and 256M keys.

Index Size: 16M
Threads: 12
Software Pipelining: 8 queries per thread

5.1. Search Performance

Multi-Threaded

- Throughput can be improved by interleaving multiple tree traversals using software pipelining
- FAST benefits most from software pipelining(x2.5), but it has relatively large latencies for comparisons
- ART performs less calculation
- It shows how various index structures utilize optimization techniques such as pipelining

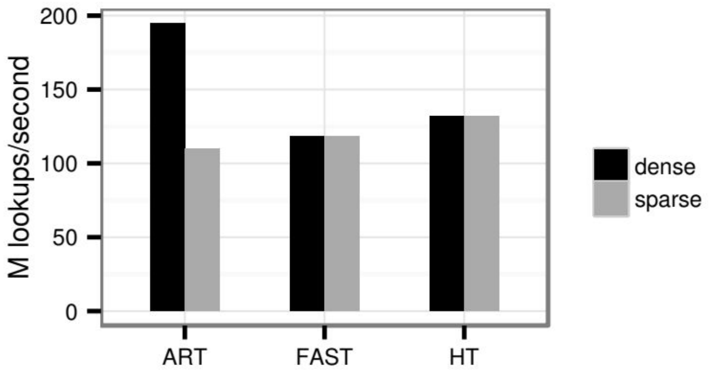


Fig. 11. Multi-threaded lookup throughput in an index with 16M keys (12 threads, software pipelining with 8 queries per thread).

5.2. Caching Effects

Index Size: 16M
Threads: 12
Operation: Random Lookup (Zipf)

Caching Effects

- Tree structures benefit from caches because top level nodes are accessed frequently
- Random lookup performs the worst for caches because of low temporal locality
- Hash table is mostly unaffected, as it does not use caches effectively
- While the trees improves with increasing cache size, because more often traversed paths are cached

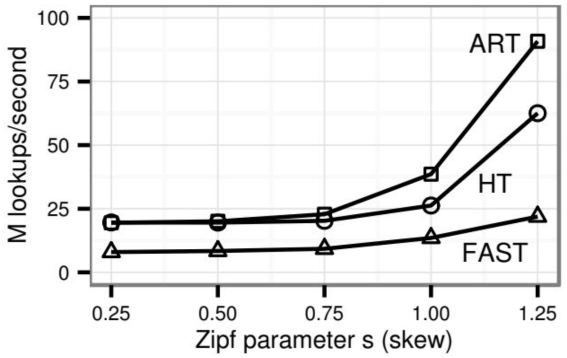


Fig. 12. Impact of skew on search performance (16M keys).

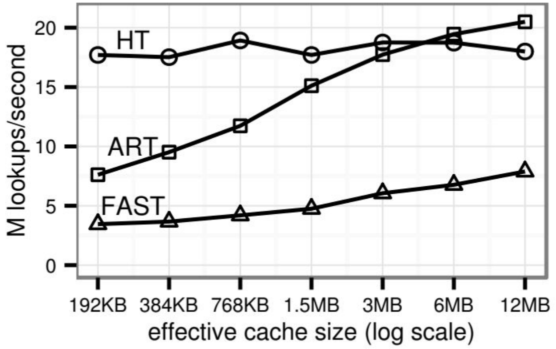


Fig. 13. Impact of cache size on search performance (16M keys).

Operation: Random Insert 16M

5.3. Update Performance

Update Performance

- ART efficiently manages key distribution within a node and saves memory for consecutive key values (Dense)
- Conversely, in a sparse, large gaps between key values may cause many nodes to have pointers that are not needed, resulting in wasted space
- ART can increase space efficiency by reducing unnecessary nodes through adaptive nodes

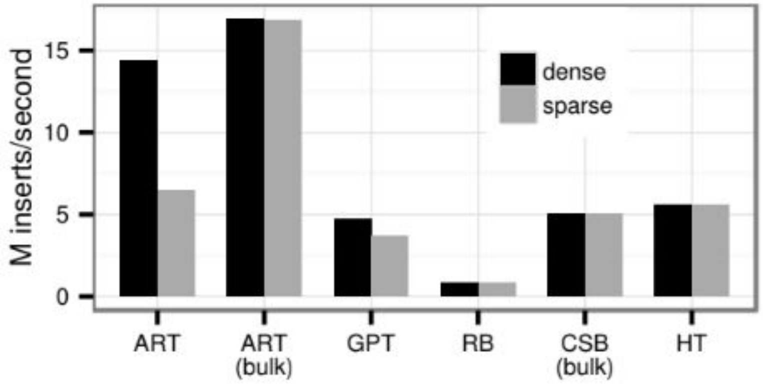


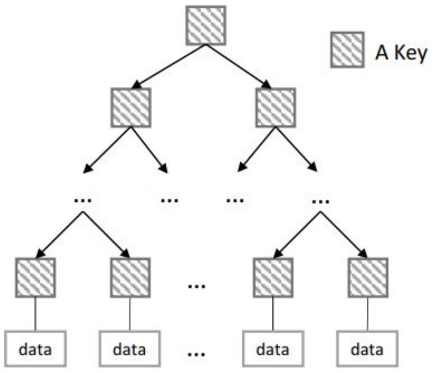
Fig. 14. Insertion of 16M keys into an empty index structure.

Operation: Random Insert 16M

5.3. Update Performance

Update Performance (FAST)

- The FAST structure is a cache-sensitive hierarchical tree structure designed to enable fast retrieval of data
- FAST+Δ includes a cyclical merge phase with O(n) complexity, which can introduce a lot of overhead and lead to poor performance when large updates occur



(a) FAST

A Comprehensive Performance Evaluation of Modern in-Memory Indices, 2018 IEEE

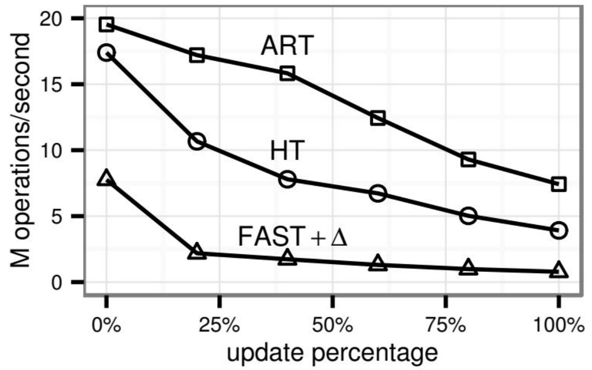


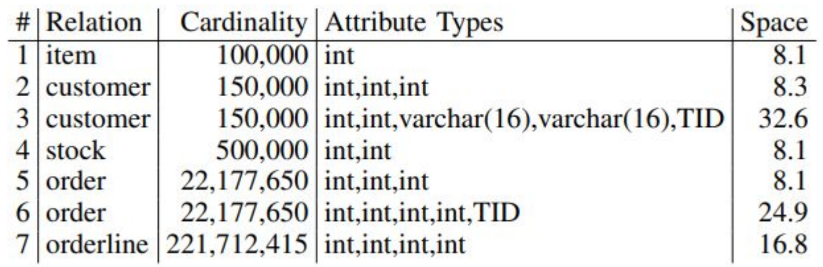
Fig. 15. Mix of lookups, insertions, and deletions (16M keys).

5.4. Impact of Collapsing Inner Nodes

Space Consumption

- The height of index 3 would be 40 without any optimizations
- Path compression and lazy expansions are critical for achieving high performance and small memory consumption with radix trees (Except for short integers)

TABLE IV
MAJOR TPC-C INDEXES AND SPACE CONSUMPTION PER KEY USING ART.



#	Relation	Cardinality	Attribute Types	Space
1	item	100,000	int	8.1
2	customer	150,000	int,int,int	8.3
3	customer	150,000	int,int,varchar(16),varchar(16),TID	32.6
4	stock	500,000	int,int	8.1
5	order	22,177,650	int,int,int	8.1
6	order	22,177,650	int,int,int,int,TID	24.9
7	orderline	221,712,415	int,int,int,int	16.8

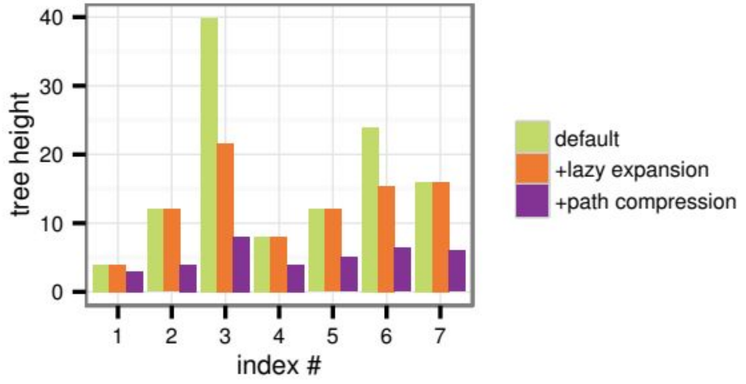


Fig. 17. Impact of lazy expansion and path compression on the height of the TPC-C indexes.

6. Conclusion

Conclusion

- As main memory developed, the existing index structure became inefficient for main memory database systems
- A high fanout, path compression, and lazy expansion reduce the tree height, and therefore lead to excellent performance
- ART was compared with the latest main memory data structure, and performance was improved and it was proven to be an excellent alternative to the existing index structure

Q&A

Thank you!

1. Background

Why ART presented

- As main memory capacity has increased, storing data in RAM has become less of a challenge

- The bottleneck in in-memory databases is index performance

- Traditional tree-based indexes are not well suited for modern hardware (e.g., cache)

- Hash is also fast, but only supports non-ordered operations (e.g., point query)

3.3. Structure of Leaf Nodes

For Non-Unique Indexes

- Non-unique key indexes have to implement additional identifier to each key

- Values can be stored in different ways
  - **Single-value leaves**
    Use additional leaf node which stores one value
    It is most general method, but has overhead when tree height increased

  - **Multi-value leaves**
    Use additional node like other inner(one of four) node, but contain values instead of pointers
    Avoid additional overhead, but require all keys in a tree to have the same length

  - **Combined pointer/value slots**
    Allow to store keys of varying length

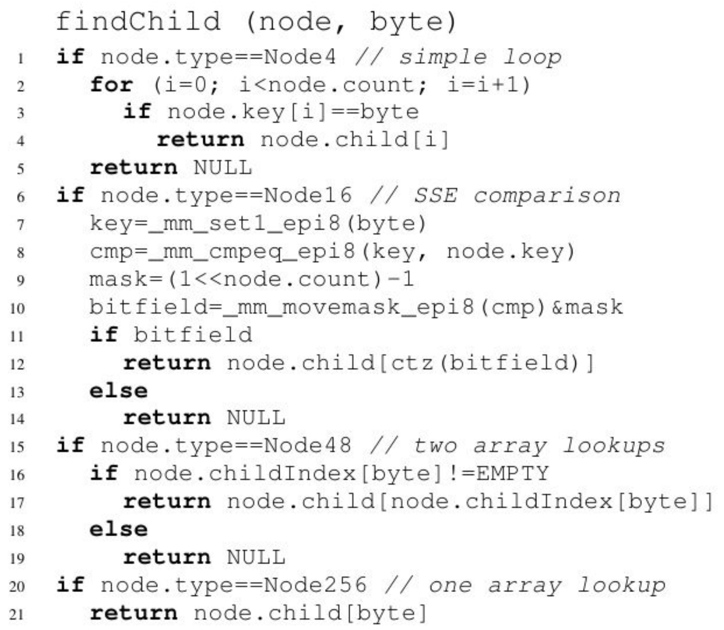
4.3. Algorithms

Search

- Traverse tree by using successive bytes of the key array

- For Node16, use SIMD to compare 16 keys with one instruction in parallel

- For Node48, check first-array if child index is valid



```
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]
```

Fig. 8. Algorithm for finding a child in an inner node given a partial key.

6. Conclusion

Conclusion

- As memory sizes have grown, index performance has become the bottleneck in in-memory databases. So ART is presented to **well suit to modern hardware architectures** and provide **general-purpose indexing structure** compared to traditional index

- Present a fast and space-efficient indexing structure for main-memory databases with high fan out, path compression, lazy expansion

-