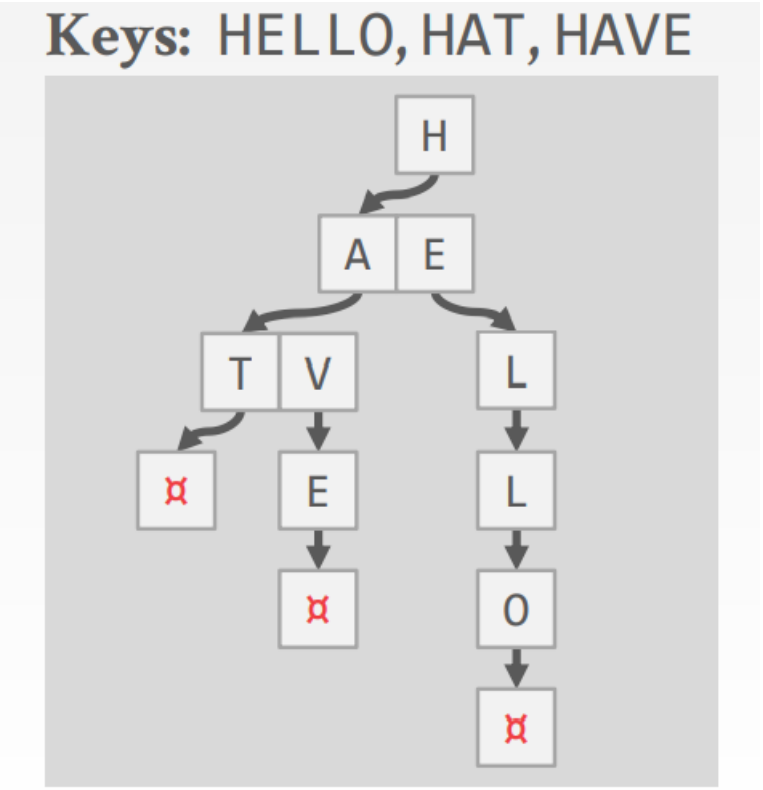
The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases

Conference: ICDE 2013
Authors: Viktor Leis, Alfons Kemper, Thomas Neumann
University: Technische Universität München

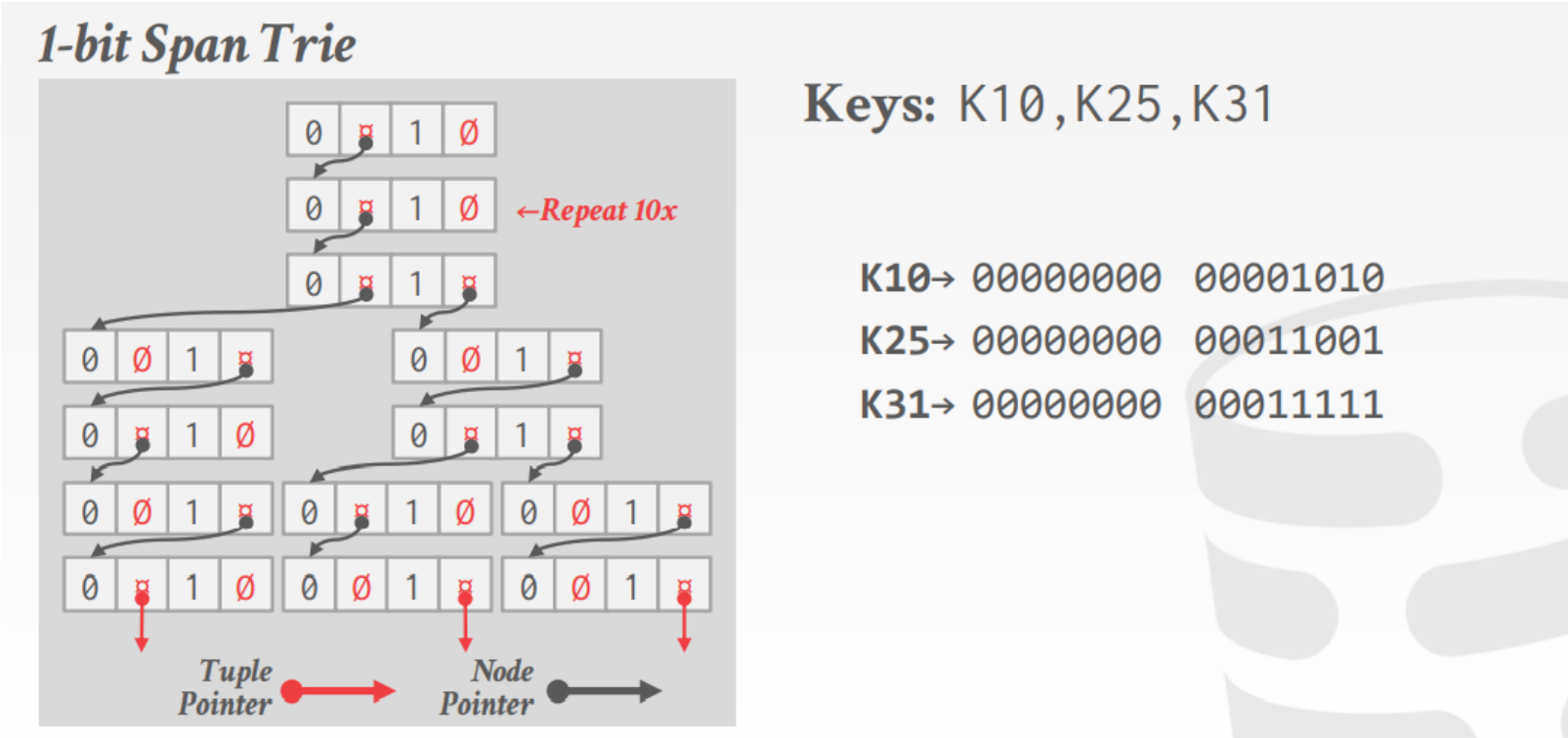
Background

- Trie Index (Also known as Digital Search Tree, Prefix Tree)
  - Shape only depends on key space and lengths.
    - Does not depend on existing keys or insertion order.
    - Does not require rebalancing operations.
  - All operations have O(k) complexity where k is the length of the key.
    - The path to a leaf node represents the key of the leaf.
    - Keys are stored implicitly and can be reconstructed from paths.



Background

- Trie Key Span
  - The span of a trie level is the number of bits that each partial key / digit represents.
    - If the digit exists in the corpus, then store a pointer to the next level in the trie branch. Otherwise, store null.
  - This determines the fan-out of each node and the physical height of the tree.
    - n-way Trie = Fan-Out of n



Related Work

- Cache Sensitive B+tree (CSB+tree)
  - Node group stored contiguously
- Two read-only search structures (pointer-free)
  - K ary search: SIMD
  - FAST: SIMD、cache、TLB

Motivation

- Radix trees have smaller height than binary search trees for $n > 2^{k/s}$.
- Radix trees, in particular with a large span, can be more efficient than traditional search trees.

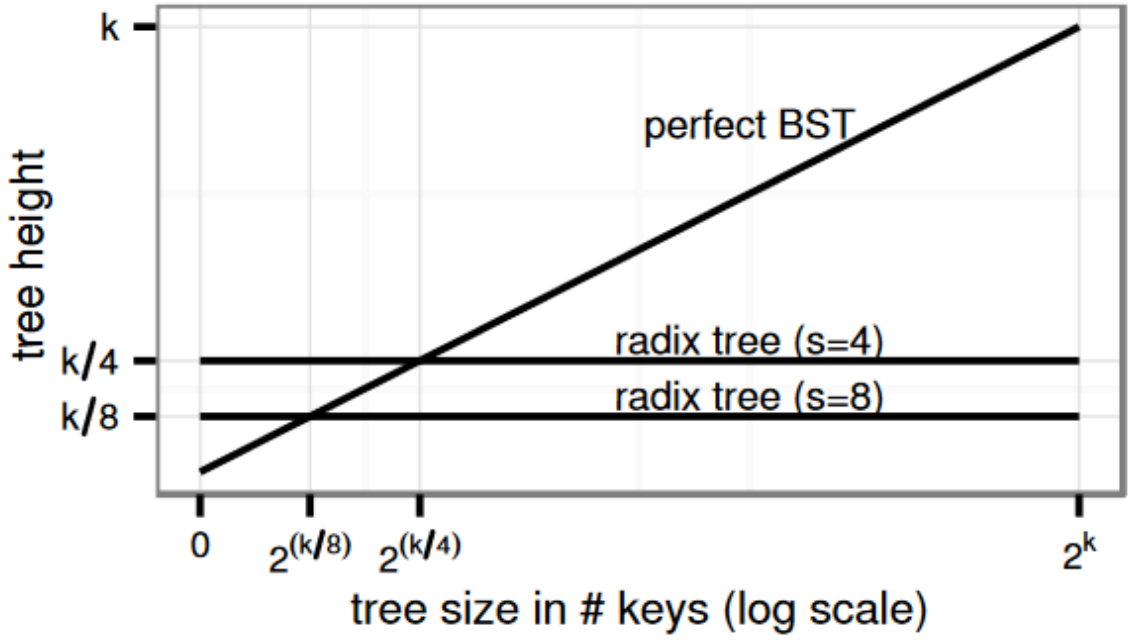


Fig. 2. Tree height of perfectly balanced binary search trees and radix trees.

Motivation

- As the span increases, the tree height decreases linearly, while the space consumption increases exponentially.

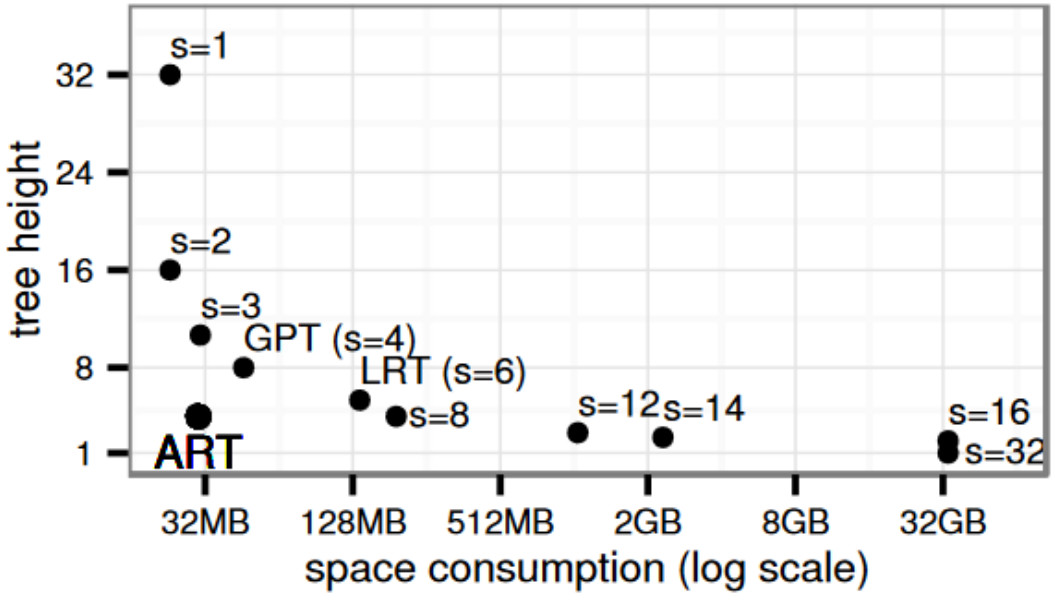


Fig. 3. Tree height and space consumption for different values of the span parameter s when storing 1M uniformly distributed 32 bit integers. Pointers are 8 byte long and nodes are expanded lazily.

Solution

- The key idea that achieves both space and time efficiency is to adaptively use different node sizes with the same, relatively large span, but with different fanout.

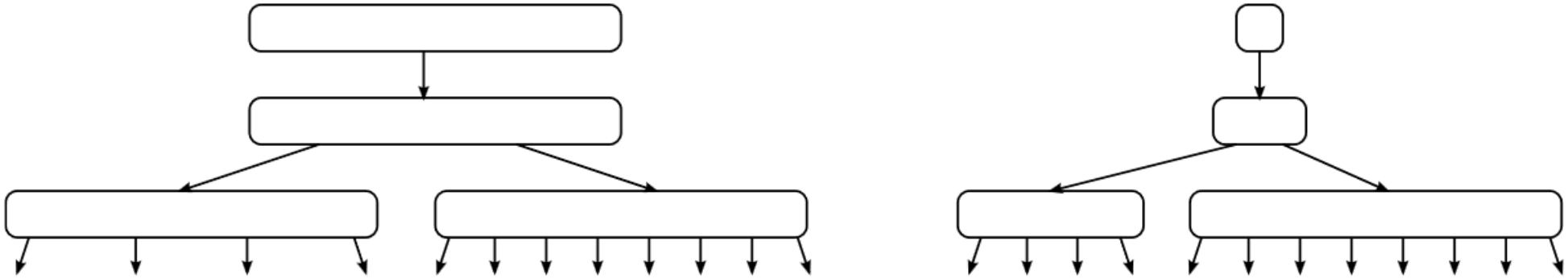


Fig. 4. Illustration of a radix tree using array nodes (left) and our adaptive radix tree ART (right).

Structure of Inner Nodes

- Four data structures
  - Node 4
  - Node 16
  - Node 48
  - Node 256
- Span = 8 bits
  - Bytes are directly addressable which avoids bit shifting and masking operations
- Split the list into one key part and one pointer part
  - Cache friendly

Structure of Inner Nodes

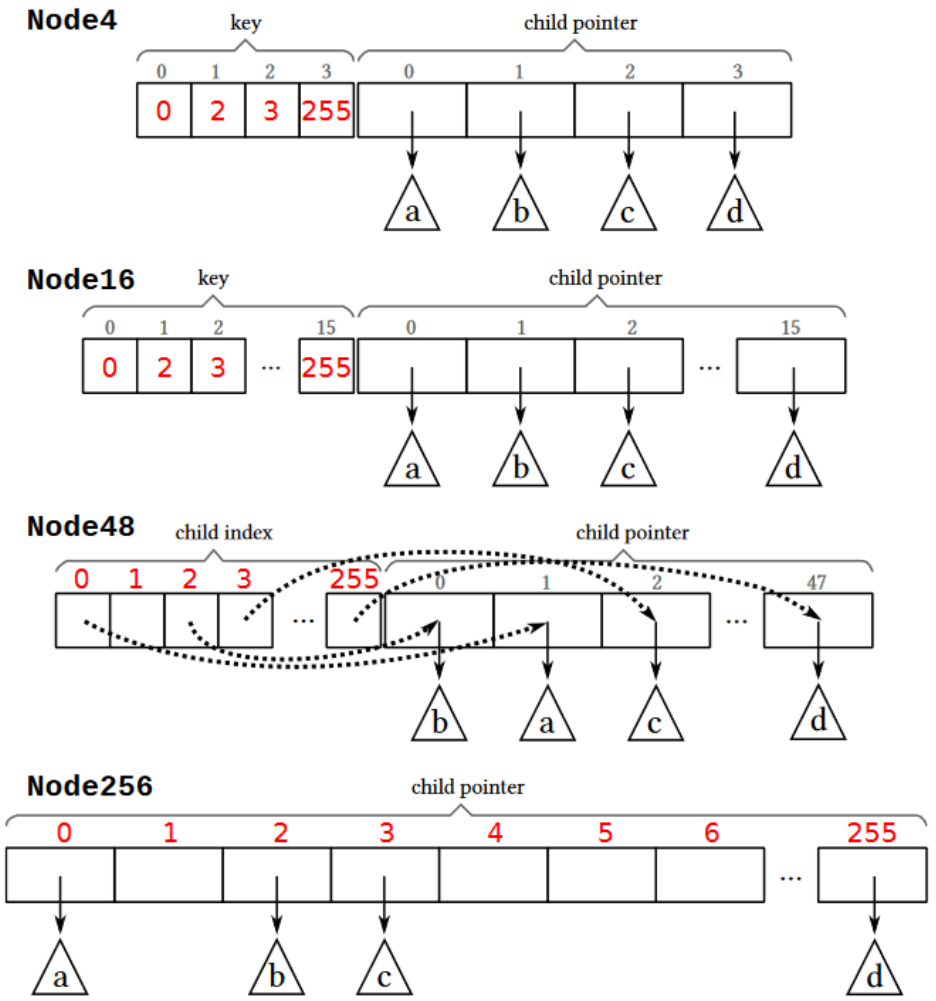


Fig. 5. Data structures for inner nodes. In each case the partial keys 0, 2, 3, and 255 are mapped to the subtrees a, b, c, and d, respectively.

Structure of Leaf Nodes

- Single-value leaves
  - The values are stored using an additional leaf node type which stores one value.
- Multi-value leaves
  - The values are stored in one of four different leaf node types, which mirror the structure of inner nodes, but contain values instead of pointers.
- Combined pointer/value slots
  - If values fit into pointers, no separate node types are necessary. Instead, each pointer storage location in an inner node can either store a pointer or a value. Values and pointers can be distinguished using one additional bit per pointer or with pointer tagging.

Collapsing Inner Nodes

- Lazy expansion
  - Inner nodes created only required to distinguish at least two leaf nodes
  - Requiring that the key is stored at the leaf or can be retrieved from the database

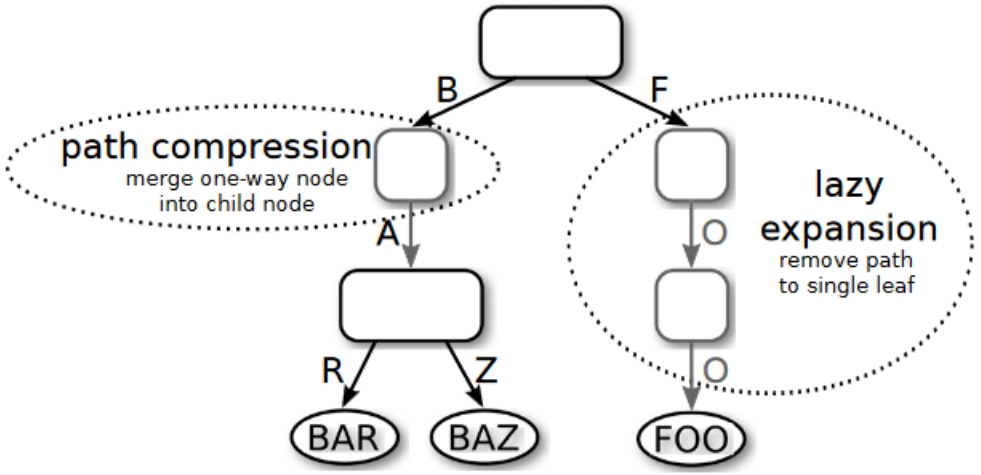


Fig. 6. Illustration of lazy expansion and path compression.

Collapsing Inner Nodes

- Path compression
  - **Pessimistic**: Collapsed prefix key stored in each node as variable length key
  - **Optimistic**: Skip collapsed partial key. Verify with the real key at the leaf
  - **Hybrid approach**: Store up to a constant-size collapsed key (8 bytes); once exceeded, switch to optimistic strategy

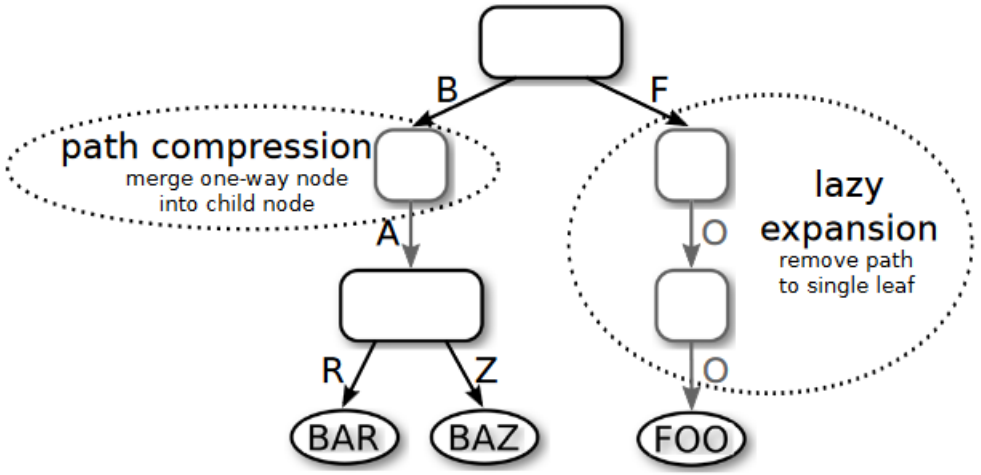


Fig. 6. Illustration of lazy expansion and path compression.

Algorithms

- Search

```
search (node, key, depth)
1  if node==NULL
2    return NULL
3  if isLeaf(node)
4    if leafMatches(node, key, depth)
5      return node
6    return NULL
7  if checkPrefix(node,key,depth)!=node.prefixLen
8    return NULL
9  depth=depth+node.prefixLen
10 next=findChild(node, key[depth])
11 return search(next, key, depth+1)
```

Fig. 7. Search algorithm.

```
findChild (node, byte)
1  if node.type==Node4 // simple loop
2    for (i=0; i<node.count; i=i+1)
3      if node.key[i]==byte
4        return node.child[i]
5    return NULL
6  if node.type==Node16 // SSE comparison
7    key=_mm_set1_epi8(byte)
8    cmp=_mm_cmpeq_epi8(key, node.key)
9    mask=(1<<node.count)-1
10   bitfield=_mm_movemask_epi8(cmp)&mask
11   if bitfield
12     return node.child[ctz(bitfield)]
13   else
14     return NULL
15 if node.type==Node48 // two array lookups
16   if node.childIndex[byte]!=EMPTY
17     return node.child[node.childIndex[byte]]
18   else
19     return NULL
20 if node.type==Node256 // one array lookup
21   return node.child[byte]
```

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

Algorithms

- Insert

```
insert (node, key, leaf, depth)
1  if node==NULL // handle empty tree
2    replace(node, leaf)
3    return
4  if isLeaf(node) // expand node
5    newNode=makeNode4()
6    key2=loadKey(node)
7    for (i=depth; key[i]==key2[i]; i=i+1)
8      newNode.prefix[i-depth]=key[i]
9    newNode.prefixLen=i-depth
10   depth=depth+newNode.prefixLen
11   addChild(newNode, key[depth], leaf)
12   addChild(newNode, key2[depth], node)
13   replace(node, newNode)
14   return
15 p=checkPrefix(node, key, depth)
16 if p!=node.prefixLen // prefix mismatch
17   newNode=makeNode4()
18   addChild(newNode, key[depth+p], leaf)
19   addChild(newNode, node.prefix[p], node)
20   newNode.prefixLen=p
21   memcpy(newNode.prefix, node.prefix, p)
22   node.prefixLen=node.prefixLen-(p+1)
23   memmove(node.prefix,node.prefix+p+1,node.prefixLen)
24   replace(node, newNode)
25   return
26 depth=depth+node.prefixLen
27 next=findChild(node, key[depth])
28 if next // recurse
29   insert(next, key, leaf, depth+1)
30 else // add to inner node
31   if isFull(node)
32     grow(node)
33   addChild(node, key[depth], leaf)
```

Algorithms

- Bulk load
  - Using the first byte of each key the key/value pairs are radix partitioned into 256 partitions and an inner node of the appropriate type is created.
  - Before returning that inner node, its children are created by recursively applying the bulk loading procedure for each partition using the next byte of each key.
- Delete
  - The leaf is removed from an inner node, which is shrunk if necessary.
  - If that node now has only one child, it is replaced by its child and the compressed path is adjusted.

Space Consumption

- Assumption
  - 8 bytes pointer
  - 16 bytes header (storing the node type, the number of non null children and the compressed path)
  - Only consider inner nodes and ignore leaf nodes

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	$16 + 256 \cdot 8 = 2064$

Space Consumption

- Leaf node: providing x bytes
- Inner node: consuming space provided by children
- If each node of a tree has a positive budget, then that tree uses less than x bytes per key.

$$b(n) = \begin{cases} x, & \text{isLeaf}(n) \\ \left(\sum_{i \in c(n)} b(i)\right) - s(n), & \text{else.} \end{cases}$$

- b(n) >= 52 for every ART node n if x =52
  - For leaves, the statement holds trivially by definition of the budget function.
  - For inner nodes, we compute a lower bound on the budget using the induction hypothesis and the minimum number of children for each node type.
- The worst-case space consumption is 52 bytes for any adaptive radix tree.

Binary Comparable Keys

- Not all attribute types can be decomposed into binary comparable digits for a radix tree.
  - Unsigned Integers: Byte order must be flipped for little endian machines.
  - Signed Integers: Flip two' s-complement so that negative numbers are smaller than positive.
  - Floats: Classify into group (neg vs. pos, normalized vs. denormalized), then store as unsigned integer.
  - Compound: Transform each attribute separately.

A transformation function $t : D \rightarrow \{0, 1, \ldots, 255\}^k$ transforms values of a domain D to binary-comparable keys of length k if it satisfies the following equivalences ($x, y \in D$):

- $x < y \Leftrightarrow \text{memcmp}_k(t(x), t(y)) < 0$
- $x > y \Leftrightarrow \text{memcmp}_k(t(x), t(y)) > 0$
- $x = y \Leftrightarrow \text{memcmp}_k(t(x), t(y)) = 0$

Evaluation

- Environment
  - Intel Core i7 3930K CPU(6 cores, 12 threads, 3.2 GHz clock rate and 3.8 GHz turbo frequency)
  - 12 MB shared, last-level cache
  - 32 GB quad-channel DDR3-1600 RAM
  - OS: Linux 3.2 in 64 bit mode
  - Compiler: GCC 4.6
- Baseline
  - Cache-Sensitive B+-tree **[CSB]**
  - Two read-only search structures optimized for modern x86 CPUs (k-ary search tree **[kary]**, Fast Architecture Sensitive Tree **[FAST]**)
  - Radix tree (Generalized Prefix Tree **[GPT]**)
  - Red-black tree **[RB]**, chained hash table **[HT]**

Evaluation

- Search Performance

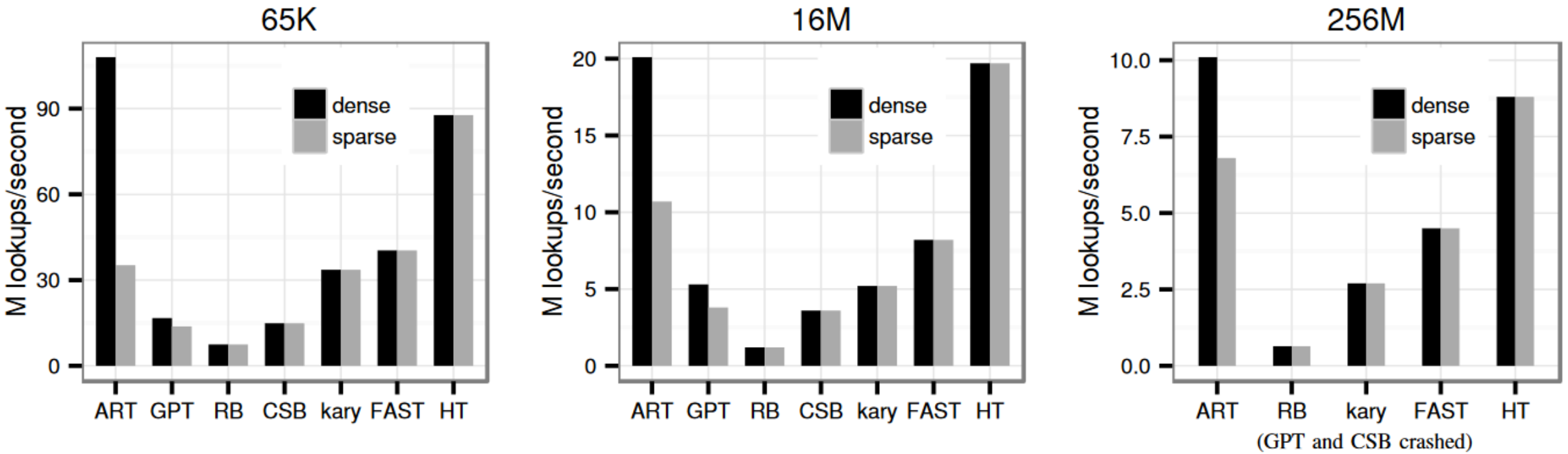


Fig. 10. Single-threaded lookup throughput in an index with 65K, 16M, and 256M keys.

Evaluation

- Search Performance

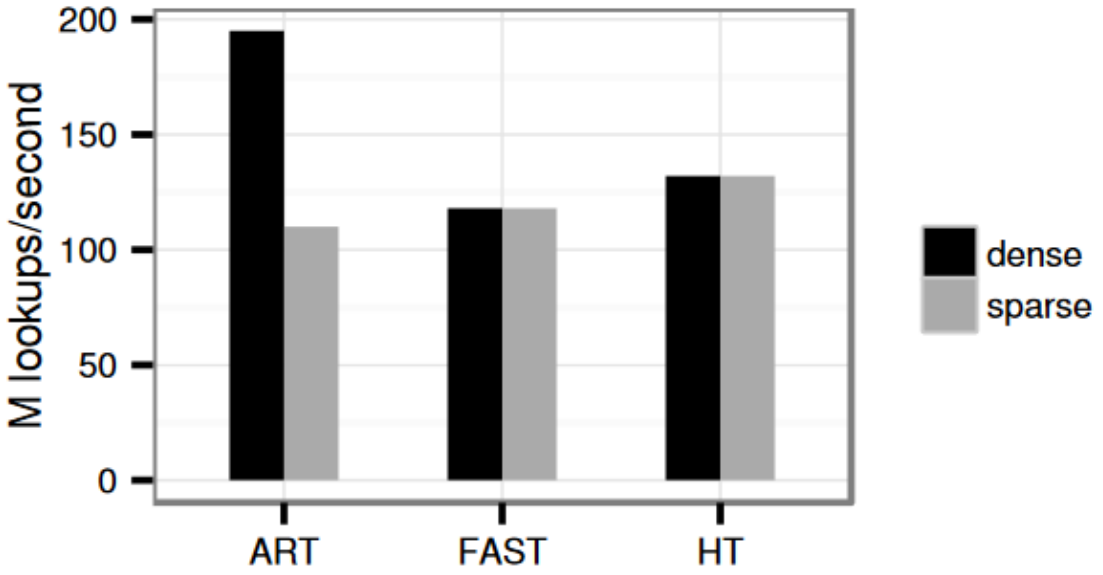


Fig. 11. Multi-threaded lookup throughput in an index with 16M keys (12 threads, software pipelining with 8 queries per thread).

Evaluation

- Caching Effects

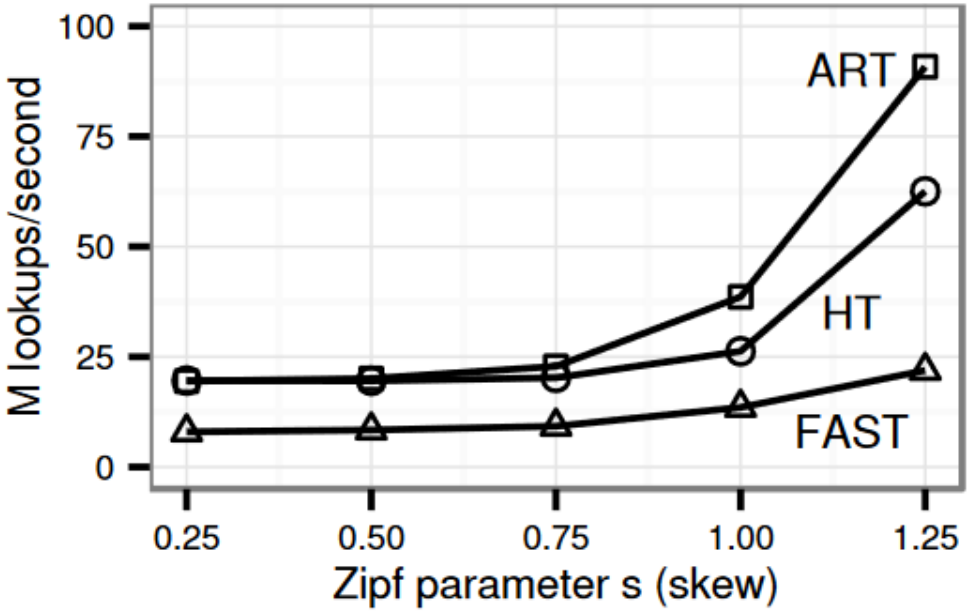


Fig. 12. Impact of skew on search performance (16M keys).

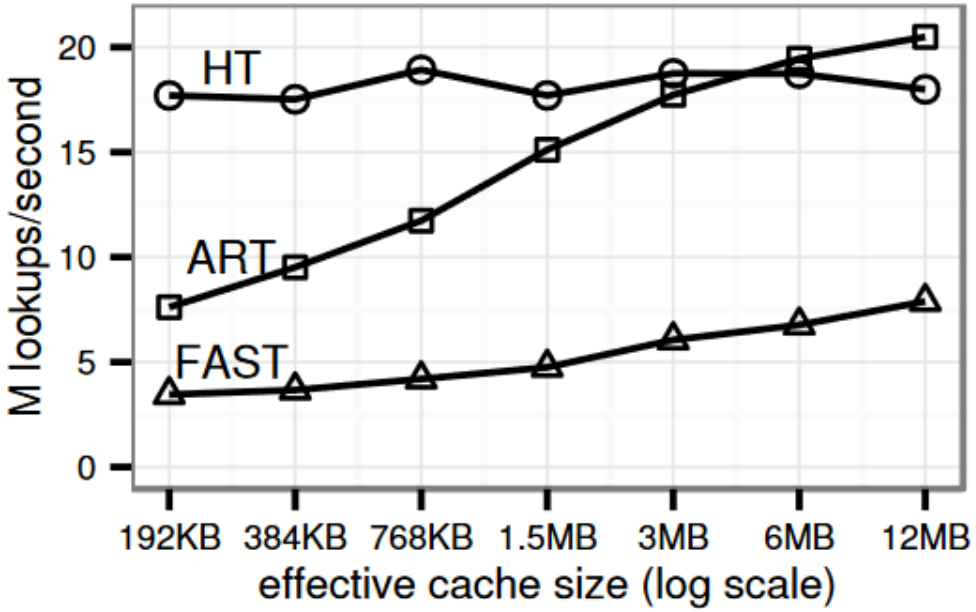


Fig. 13. Impact of cache size on search performance (16M keys).

Evaluation

- Updates

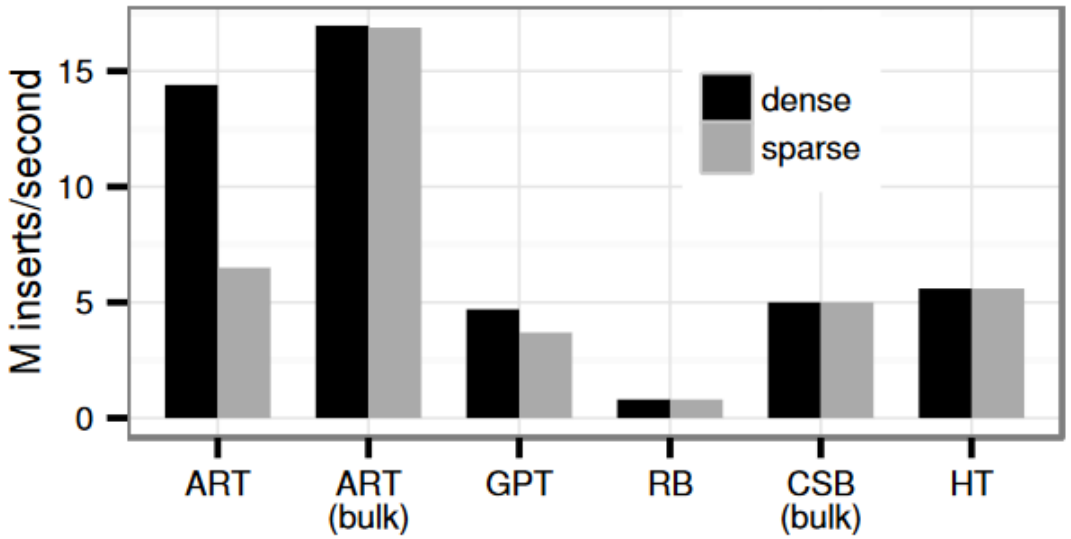


Fig. 14. Insertion of 16M keys into an empty index structure.

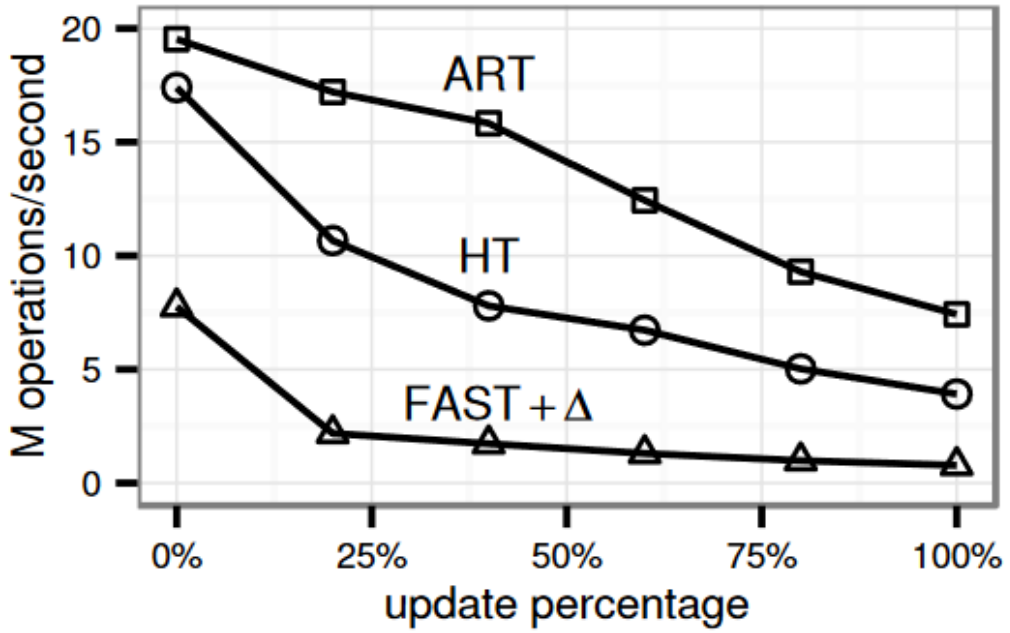


Fig. 15. Mix of lookups, insertions, and deletions (16M keys).

Evaluation

- End-to-End Evaluation

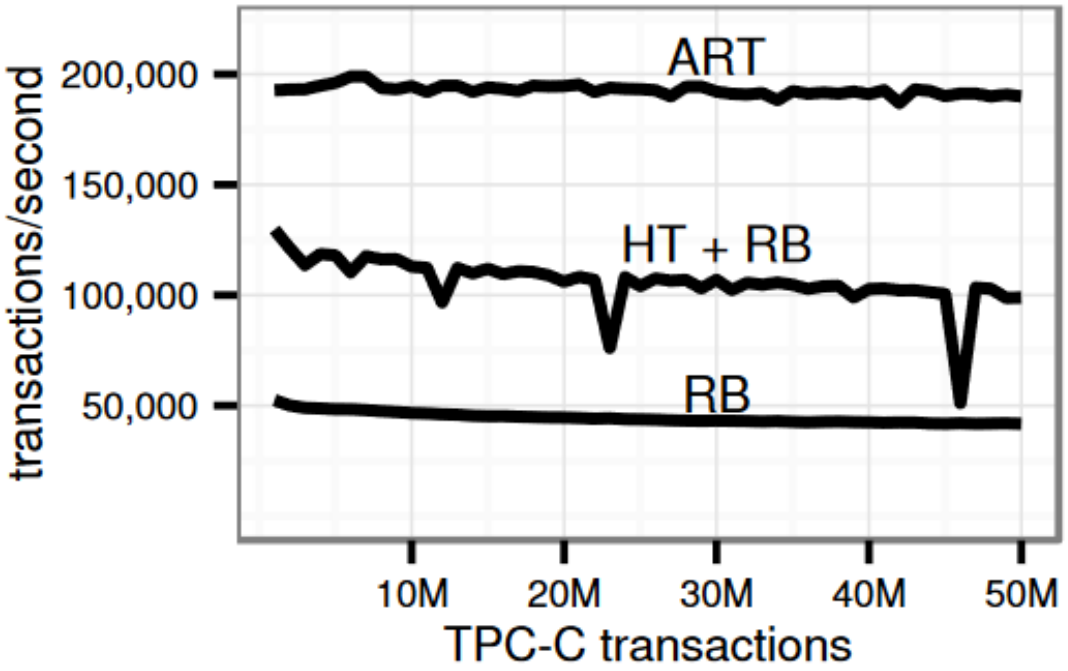


Fig. 16. TPC-C performance.

Evaluation

- End-to-End Evaluation

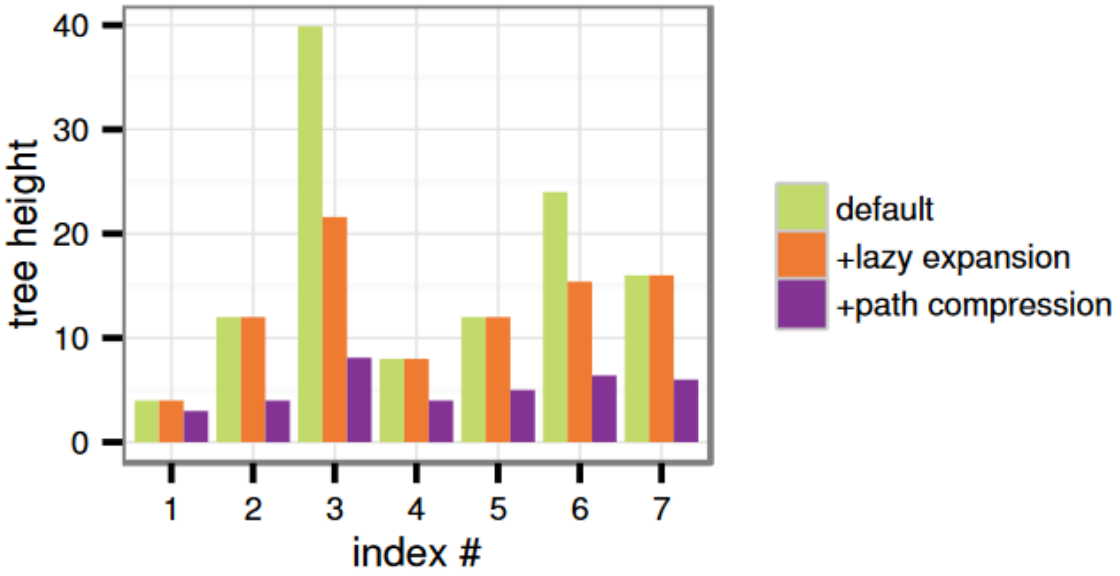


Fig. 17. Impact of lazy expansion and path compression on the height of the TPC-C indexes.

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

Thank You!