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

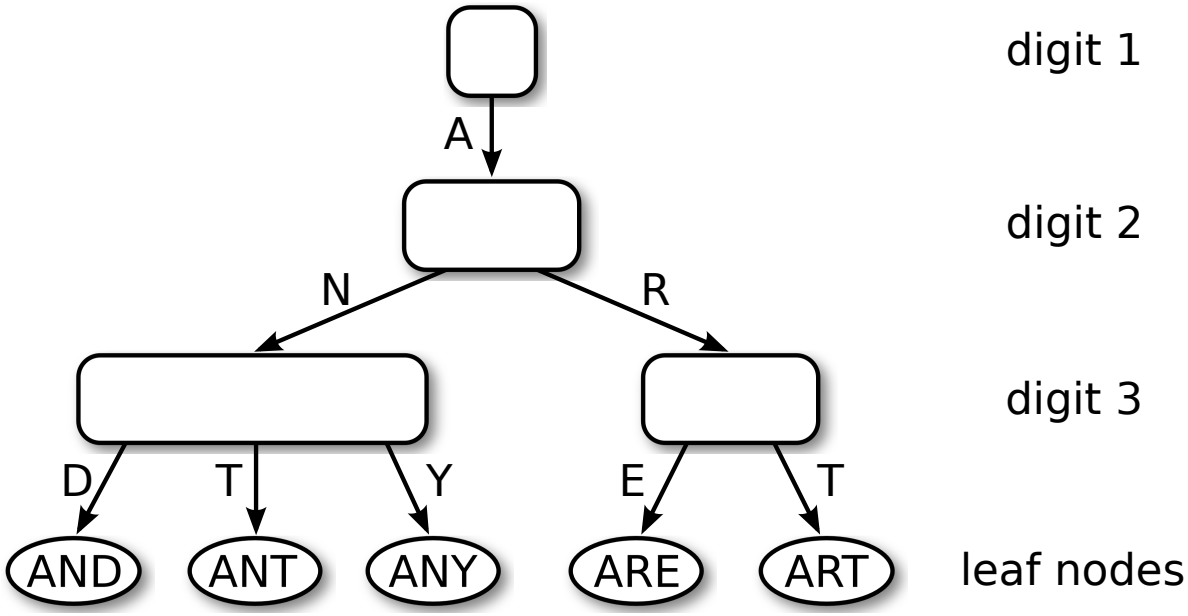
Viktor Leis, Alfons Kemper, Thomas Neumann

Technische Universität München

Introduction

- The performance of index structures is critical for query and transaction execution.
- B-Trees dominate disk-based indexes.
- In main memory, hash tables and search trees are common.
- But hash tables are unordered, and search trees are slow.
- Can we do better?

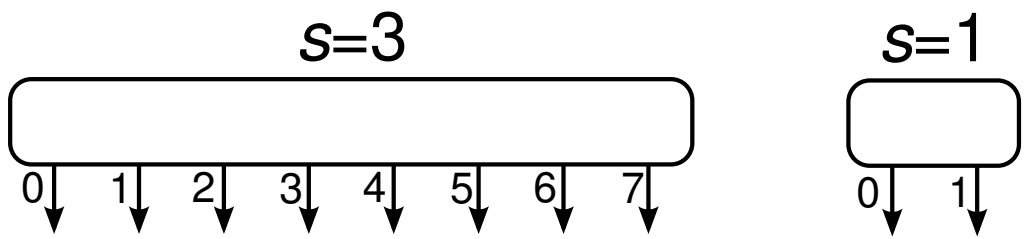
Trie, Radix Tree, Prefix Tree, Digital Tree



- Tree height depends on key length k, but not on tree size n
- No rebalancing required
- Lexicographic order

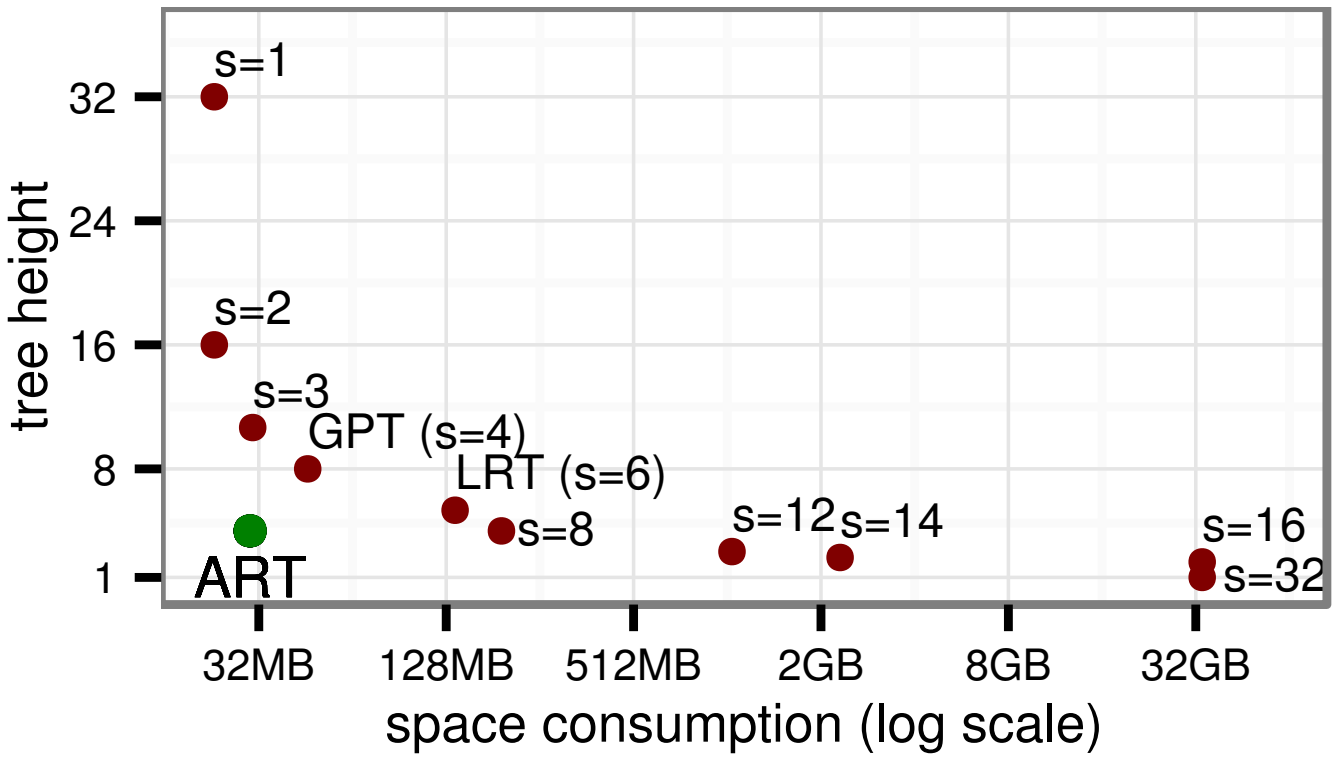
Radix Tree Span

- For binary keys, the fanout can be configured.
- At each node, s bits ("span") of the key are used.
- Each inner node is simply an array of 2^s pointers.



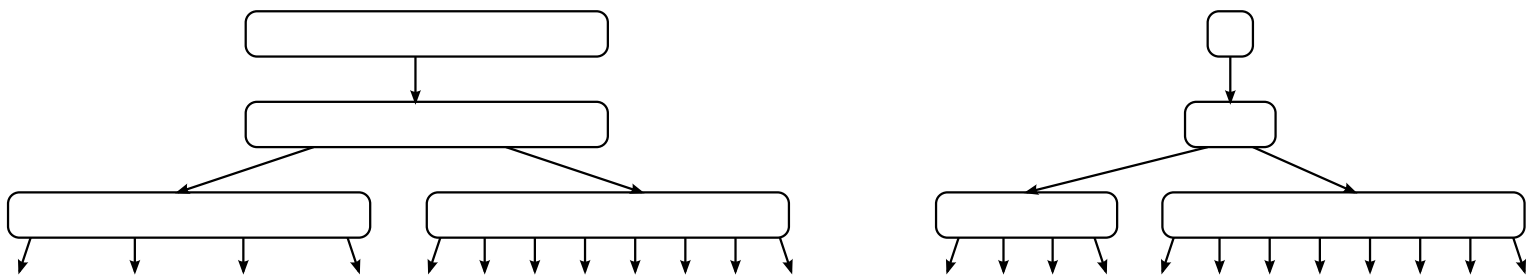
Height vs. Space Tradeoff

- Large s: Small height (fast), large space consumption
- Small s: Large height (slow), small space consumption
- ART allows to escape this tradeoff



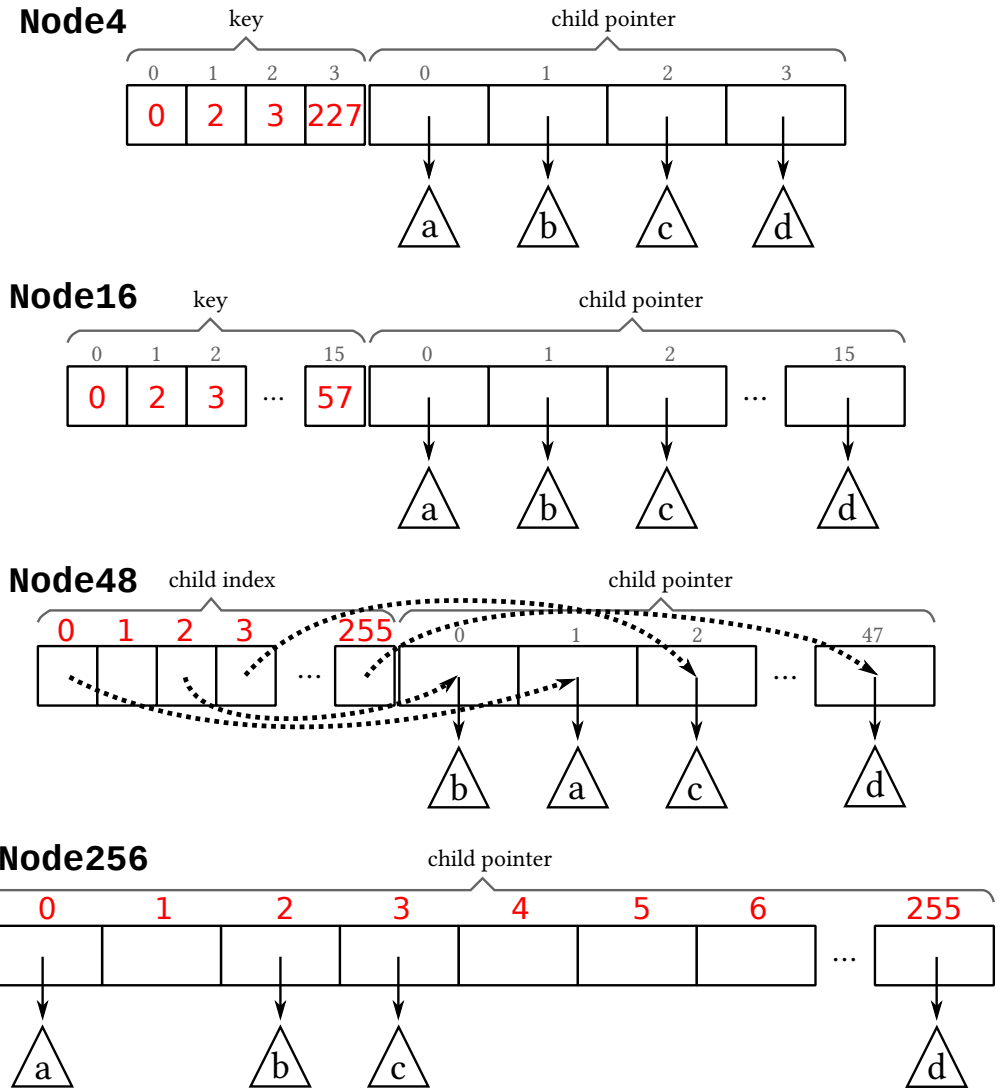
Main Idea: Adaptively Sized Nodes

- $s = 8$: each inner node maps 1 byte of the key to the next child node
- Different node sizes depending on number of children (variable fanout)

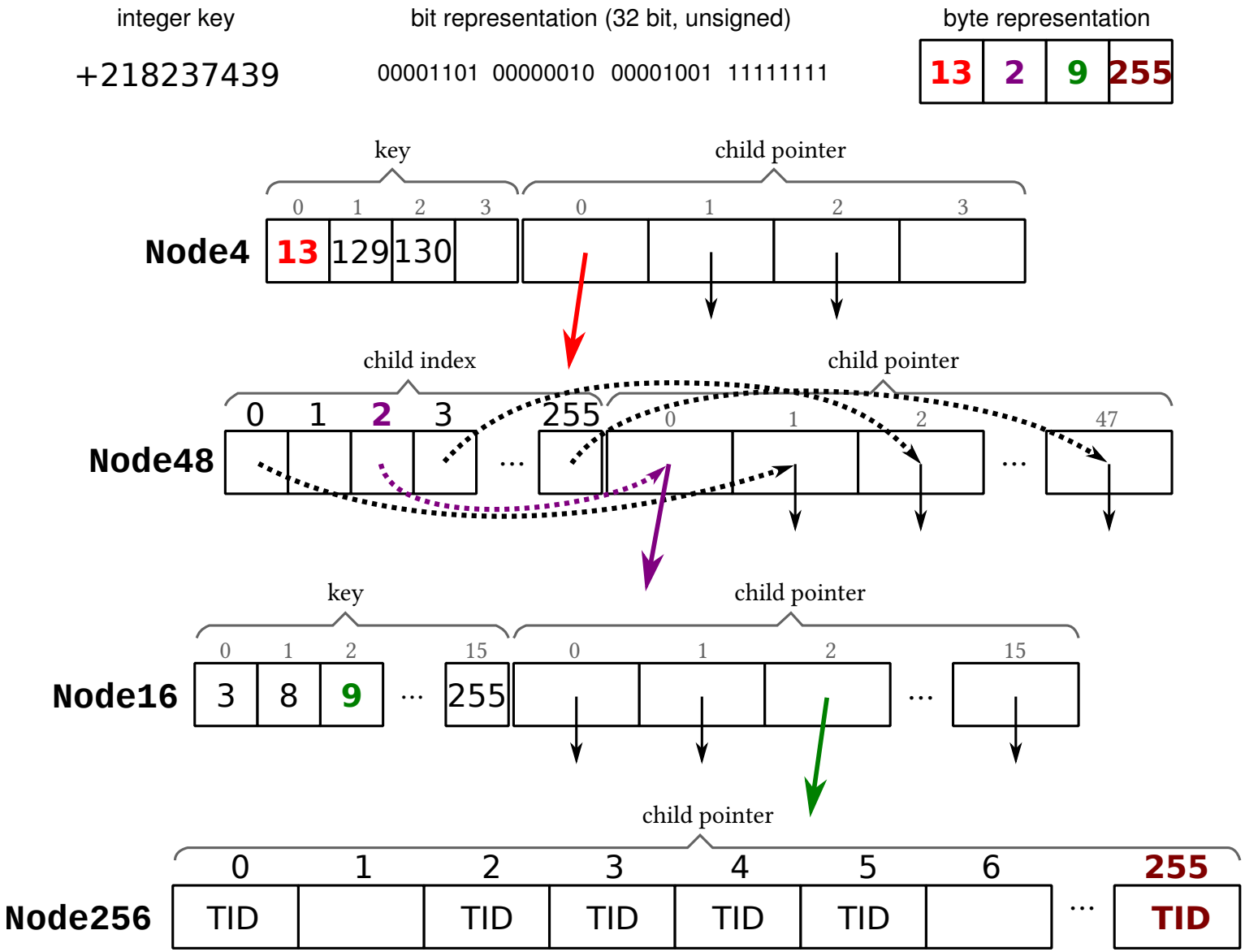


Internal Data Structures

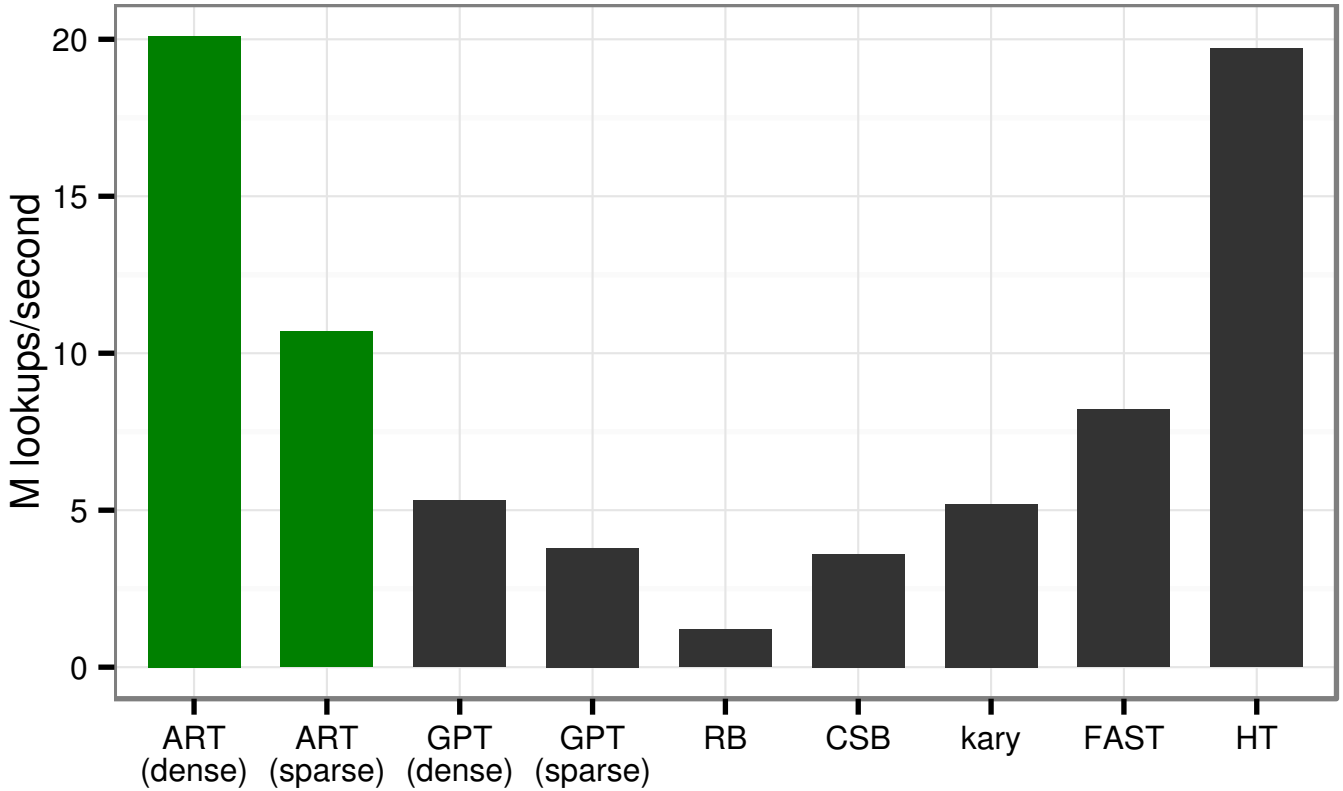
- We use 4 data structures with different capacities and dynamically chose the best one for each node



Traversal


integer key
+218237439
bit representation (32 bit, unsigned)
00001101 00000010 00001001 11111111
byte representation
13 2 9 255
key
child pointer
Node4
0 1 2 3
13 129 130
child index
child pointer
Node48
0 1 2 3 255
0 1 2 47
key
child pointer
Node16
0 1 2 15
3 8 9 255
child pointer
Node256
0 1 2 3 4 5 6 255
TID TID TID TID TID TID

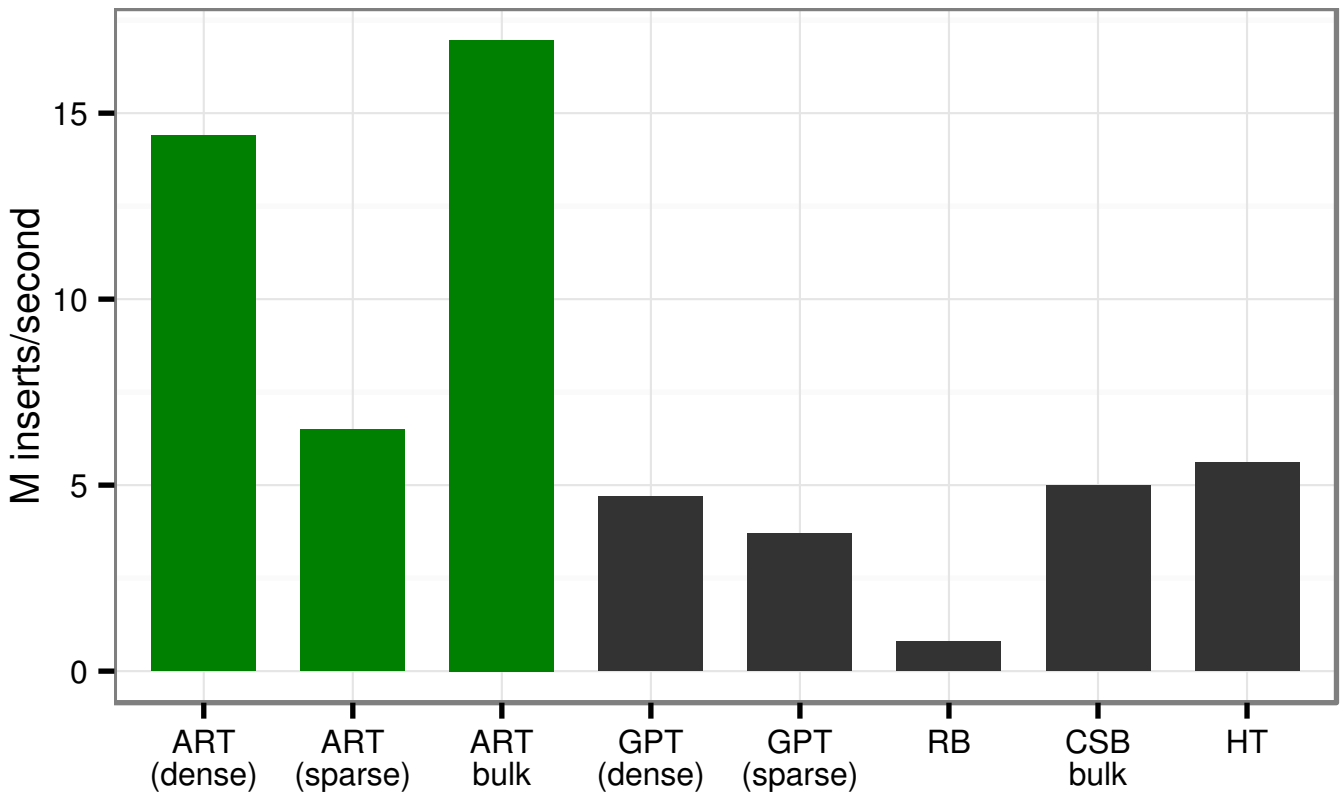
Lookup Performance (4 Byte Keys)



- GPT: Generalized Prefix Tree, Boehm et al., BTW 2011
- RB: Red-Black Tree
- CSB: Cache-Sensitive B+Tree, Rao and Ross, SIGMOD 2000
- kary: K-ary Search Tree, Schlegel et at., Damon 2009
- FAST: Fast Architecture Sensitive Tree, Kim et al., SIGMOD 2010
- HT: Chained Hash Table

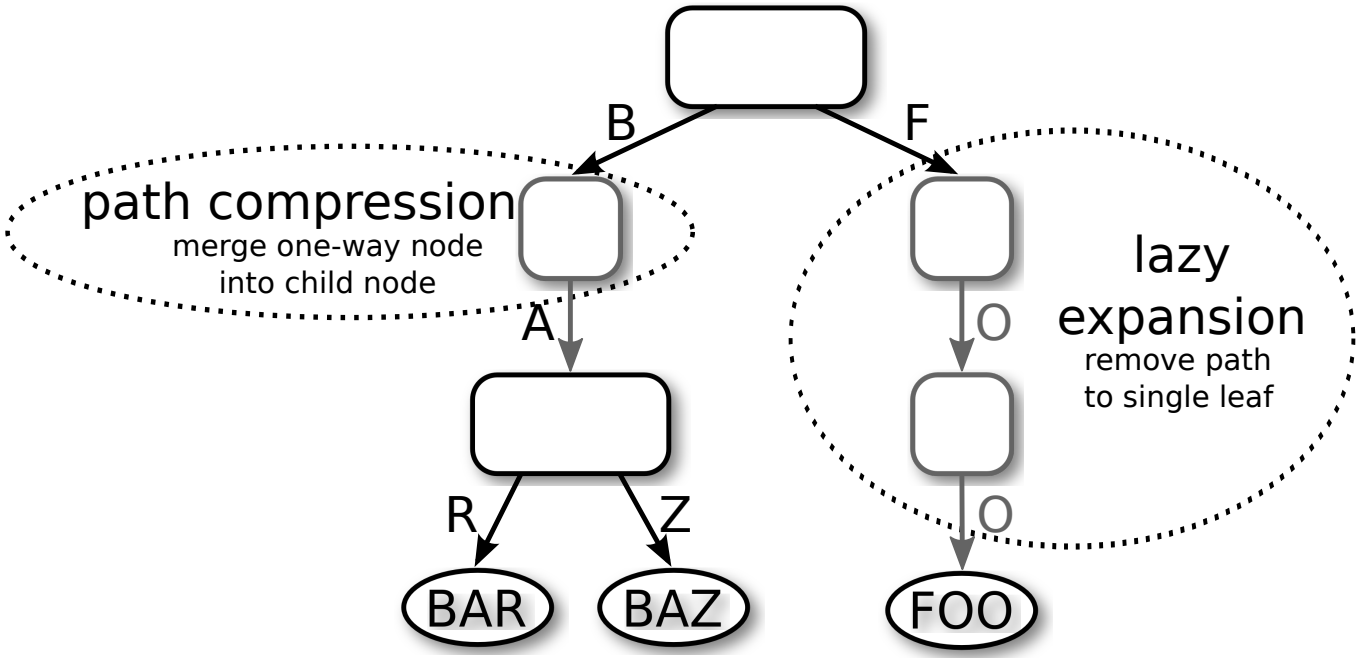
Insert Performance

- 16M entries, 4 byte keys



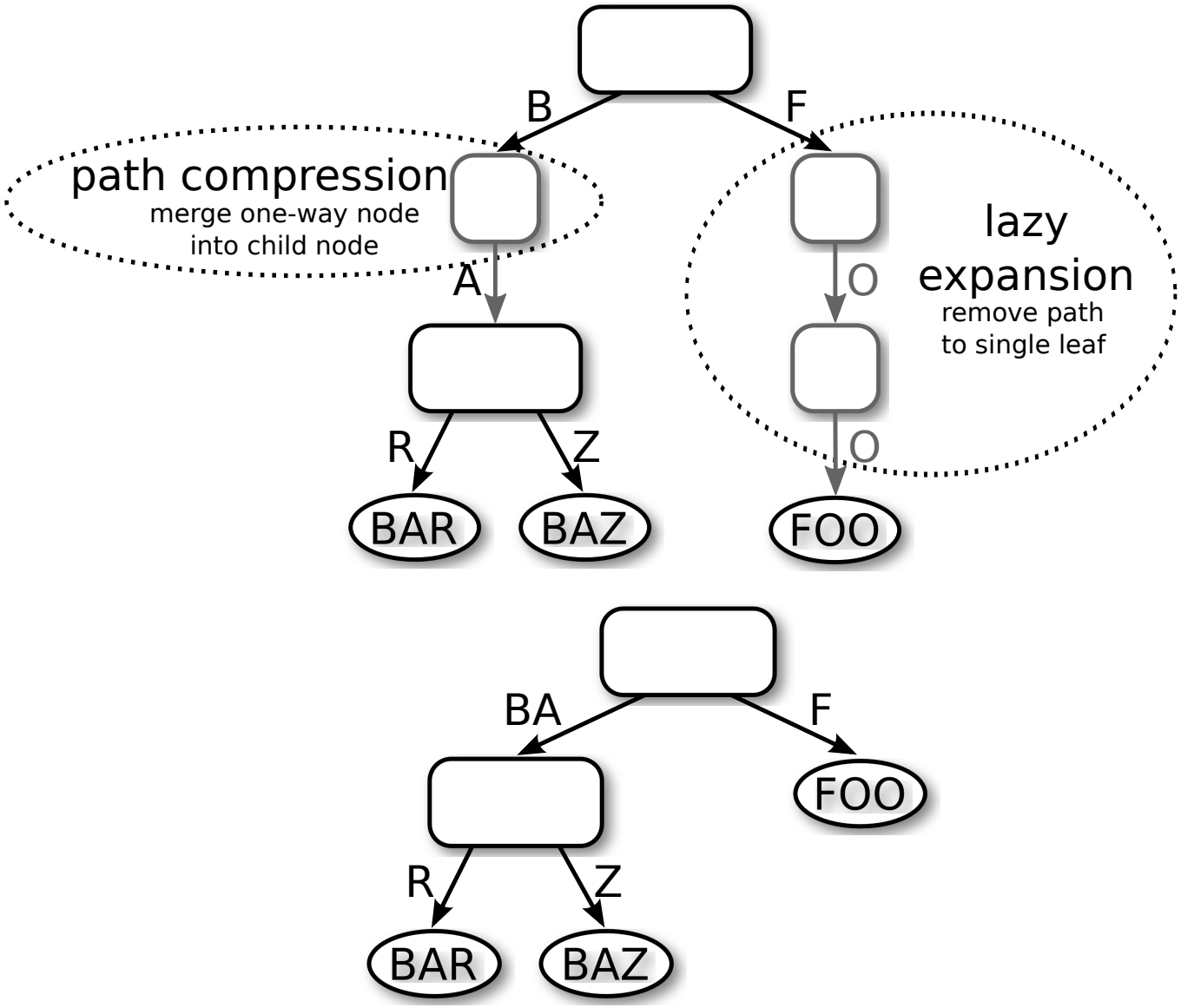
Height Optimizations for Long Keys

- Remove all one-way nodes



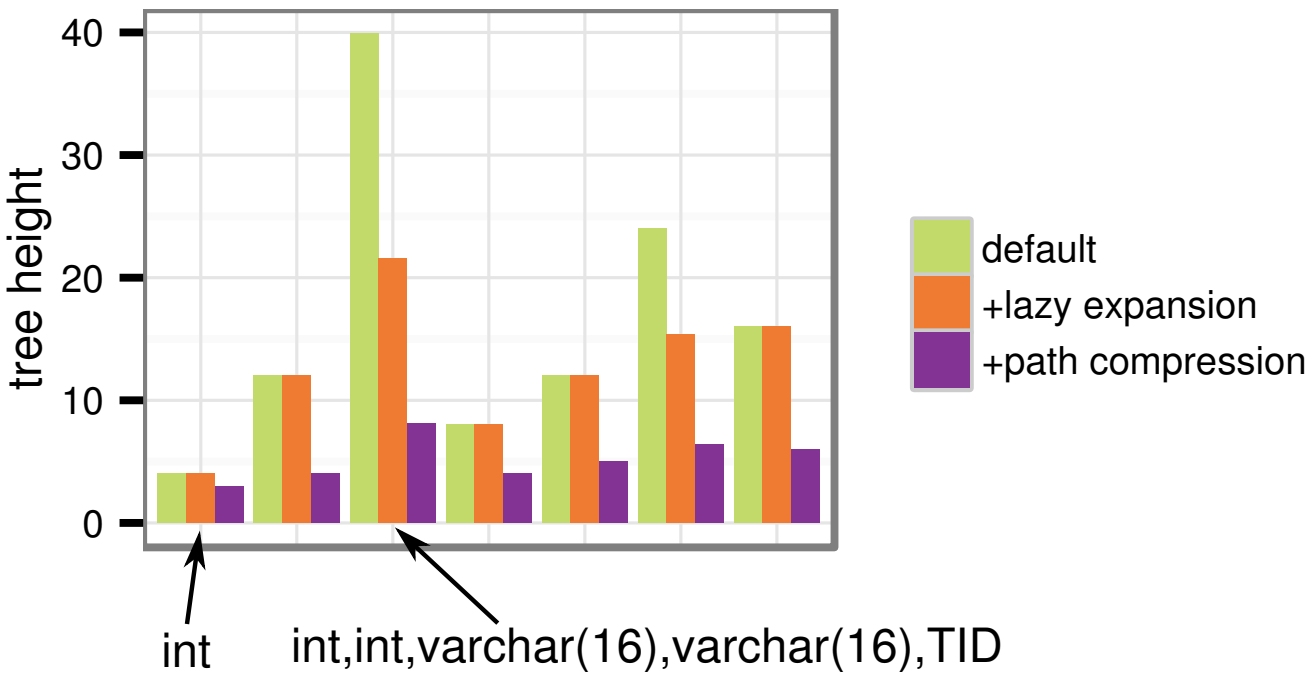
Height Optimizations for Long Keys

- Remove all one-way nodes



Effect of Height Optimizations on TPC-C indices

- Without the height optimizations the height is the length of the keys in bytes (too much for long keys)

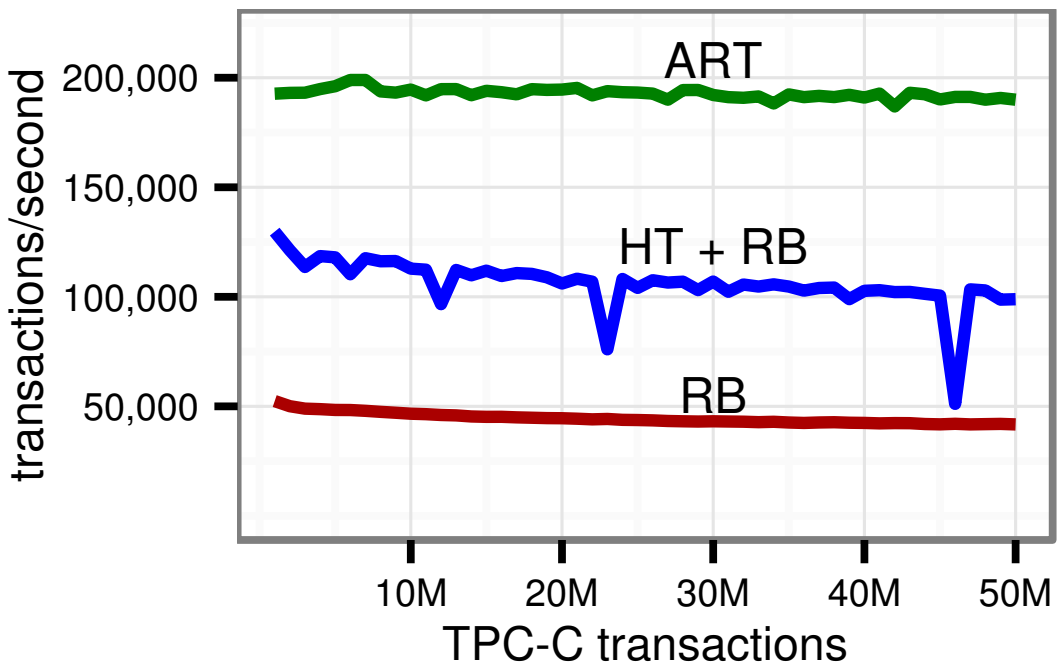


Space Consumption

- Depends on key distribution and key length
- Best case: 8.1 bytes per key for dense integers (e.g. surrogate primary keys)
- Worst case: 52 bytes (even for arbitrarily long keys, proof in paper)
- In practice much better than worst case even for relatively long strings
- Space consumption can be reduced further by adding more node types

End-to-End Performance (TPC-C)

- ART is now the default indexing structure in our Main-Memory Database System HyPer
- Uses only half the space than the hash table and the Red-Black tree, since most indexed attributes of TPC-C are dense integers.



Conclusions

ART is

- a general-purpose indexing structure which can be used for all typical built-in data types,
- is space efficient, and
- has excellent search, insert, and deletion performance.

Therefore, ART combines the benefits of search trees (ordered keys) with the efficiency of hash tables.