WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

CS 764: Topics in Database Management Systems
Lecture 15: Adaptive Radix Tree

Xiangyao Yu
10/27/2020

Today’s Paper: B-tree Locking

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

Viktor Leis, Alfons Kemper, Thomas Neumann

Fakultät für Informatik
Technische Universität München
Boltzmannstrae 3, D-85748 Garching
<lastname>@in.tum.de

***Abstract*—Main memory capacities have grown up to a point where most databases fit into RAM. For main-memory database systems, index structure performance is a critical bottleneck. Traditional in-memory data structures like balanced binary search trees are not efficient on modern hardware, because they do not optimally utilize on-CPU caches. Hash tables, also often used for main-memory indexes, are fast but only support point queries.**

To overcome these shortcomings, we present ART, an adaptive radix tree (trie) for efficient indexing in main memory. Its lookup performance surpasses highly tuned, read-only search trees, while supporting very efficient insertions and deletions as well. At the same time, ART is very space efficient and solves the problem of excessive worst-case space consumption, which plagues most radix trees, by adaptively choosing compact and efficient data structures for internal nodes. Even though ART's performance is comparable to hash tables, it maintains the data in sorted order, which enables additional operations like range scan and prefix lookup.

I. Introduction

After decades of rising main memory capacities, even large transactional databases fit into RAM. When most data is cached, traditional database systems are CPU bound because they spend considerable effort to avoid disk accesses. This has led to very intense research and commercial activities in main-memory database systems like H-Store/VoltDB [1], SAP HANA [2], and HyPer [3]. These systems are optimized for the new hardware landscape and are therefore much faster. Our system HyPer, for example, compiles transactions to machine code and gets rid of buffer management, locking, and latching overhead. For OLTP workloads, the resulting execution plans are often sequences of index operations. Therefore, index efficiency is the decisive performance factor.

More than 25 years ago, the T-tree [4] was proposed as an in-memory indexing structure. Unfortunately, the dramatic processor architecture changes have rendered T-trees, like all traditional binary search trees, inefficient on modern hardware. The reason is that the ever growing CPU cache sizes and the diverging main memory speed have made the underlying assumption of uniform memory access time obsolete. B^+-tree variants like the cache sensitive B^+-tree [5] have more cache-friendly memory access patterns, but require more expensive update operations. Furthermore, the efficiency of both binary and B^+-trees suffers from another feature of modern CPUs: Because the result of comparisons cannot be predicted easily,

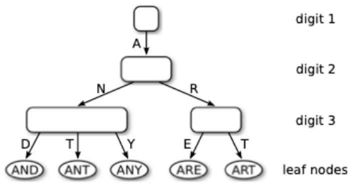


Fig. 1. Adaptively sized nodes in our radix tree.

the long pipelines of modern CPUs stall, which causes additional latencies after every second comparison (on average).

These problems of traditional search trees were tackled by recent research on data structures specifically designed to be efficient on modern hardware architectures. The k-ary search tree [6] and the Fast Architecture Sensitive Tree (FAST) [7] use data level parallelism to perform multiple comparisons simultaneously with Singe Instruction Multiple Data (SIMD) instructions. Additionally, FAST uses a data layout which avoids cache misses by optimally utilizing cache lines and the Translation Lookaside Buffer (TLB). While these optimizations improve search performance, both data structures cannot support incremental updates. For an OLTP database system which necessitates continuous insertions, updates, and deletions, an obvious solution is a differential file (delta) mechanism, which, however, will result in additional costs.

Hash tables are another popular main-memory data structure. In contrast to search trees, which have $O(\log n)$ access time, hash tables have expected $O(1)$ access time and are therefore much faster in main memory. Nevertheless, hash tables are less commonly used as database indexes. One reason is that hash tables scatter the keys randomly, and therefore only support point queries. Another problem is that most hash tables do not handle growth gracefully, but require expensive reorganization upon overflow with $O(n)$ complexity. Therefore, current systems face the unfortunate trade-off between fast hash tables that only allow point queries and fully-featured, but relatively slow, search trees.

A third class of data structures, known as trie, radix tree, prefix tree, and digital search tree, is illustrated in Figure 1.

ICDE 2013

Outline

B-tree vs. Trie

Adaptive Radix Tree

- Adaptive types
- Collapsing inner nodes
- Search and insert operations

Evaluation

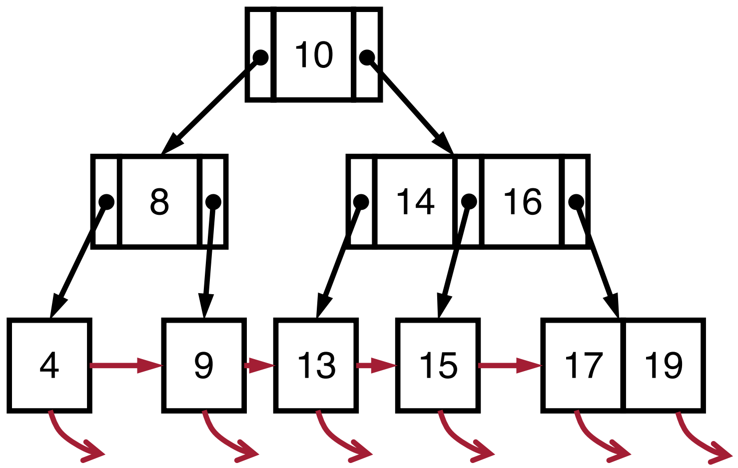
B+ Tree Revisit

Modern indexes fit in main memory

Keys are stored in each level of the tree

Must always traverse to the leaf node to check existence (e.g., cannot stop at an inner node)

B+ tree

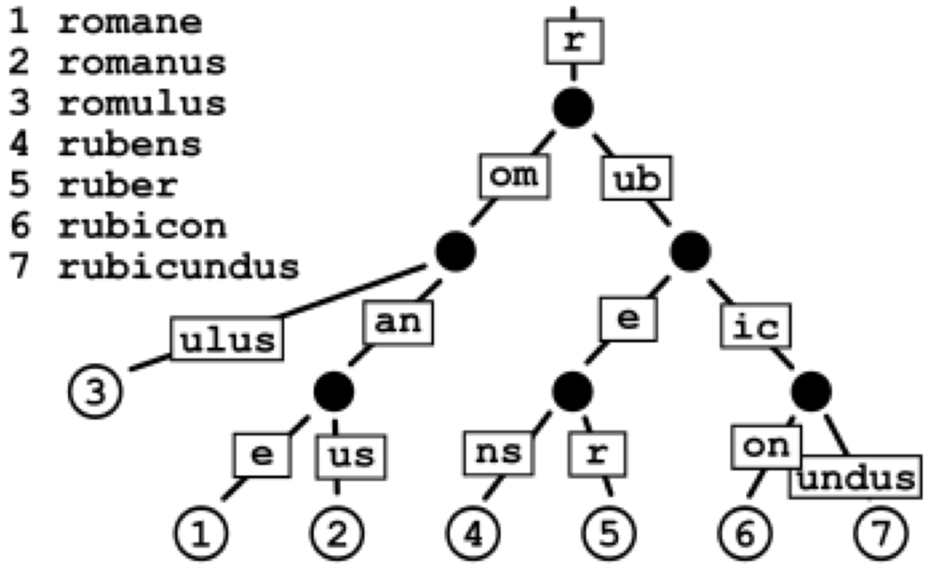


Trie (aka. digital tree or prefix tree)

Path to leaf node represents key of the leaf

Operation complexity is *O(k)* where *k* is the length of the key

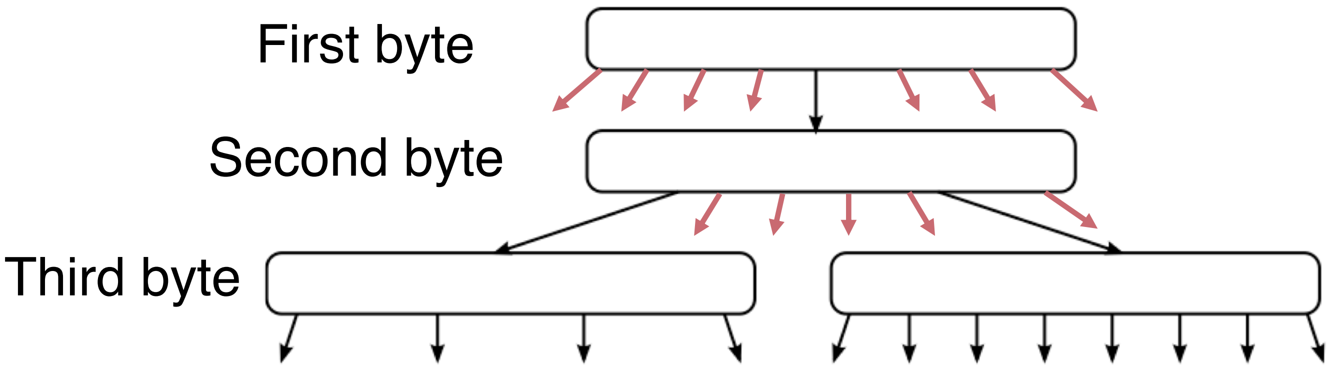
Keys are most often strings and each node contains characters



Source: https://en.wikipedia.org/wiki/Radix_tree

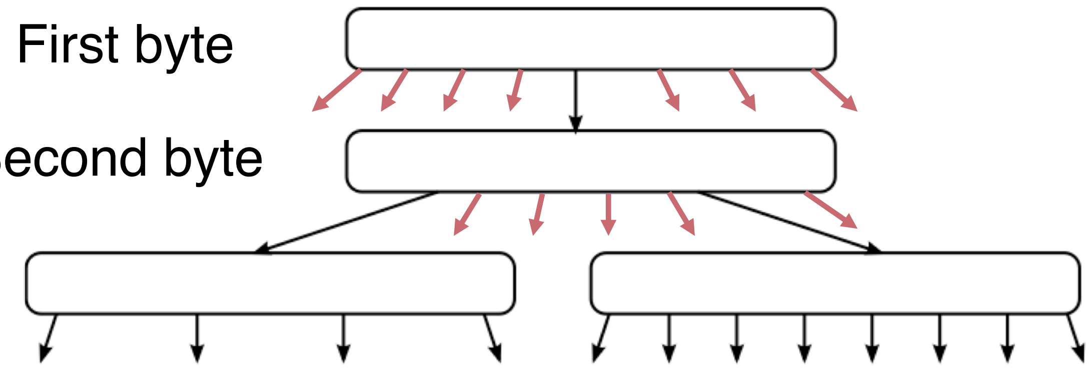
Static Radix Tree

Span: The number of bits within the key used to determine the next child



Static Radix Tree

Span: The number of bits within the key used to determine the next child



Large span

=> reduced height

=> exponential tree size

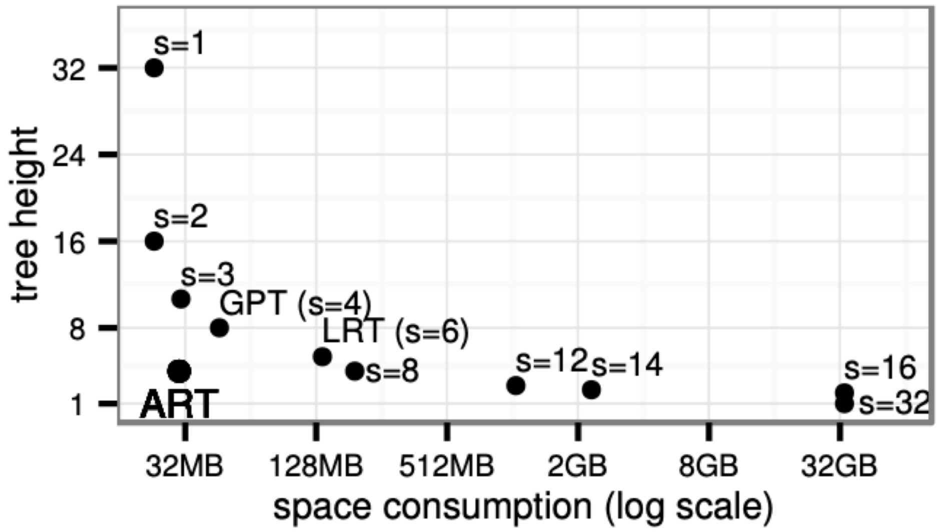
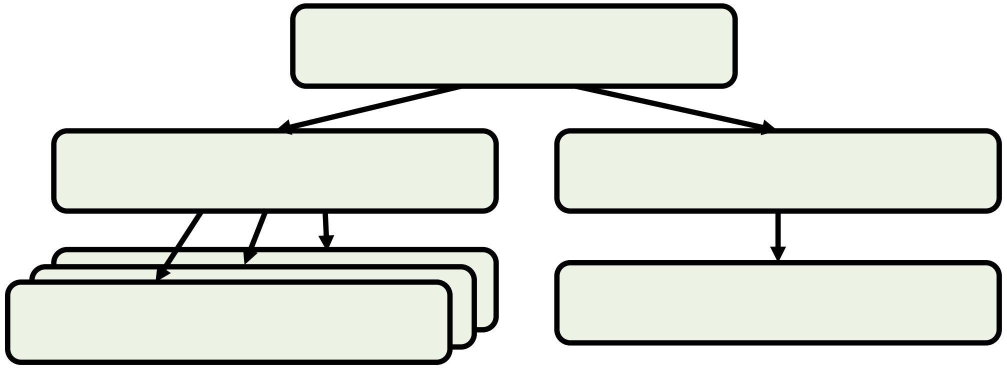


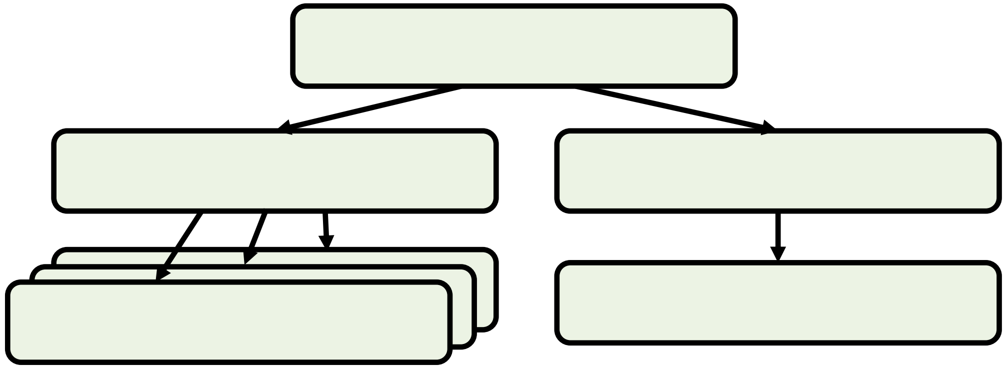
Fig. 3. Tree height and space consumption for different values of the span parameter s when storing 1M uniformly distributed 32 bit integers. Pointers are 8 byte long and nodes are expanded lazily.

Key Idea: Adaptive Radix Tree

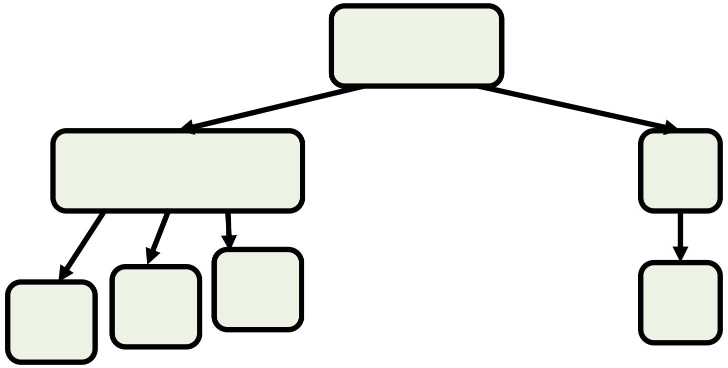


Original Radix Tree

Key Idea: Adaptive Radix Tree

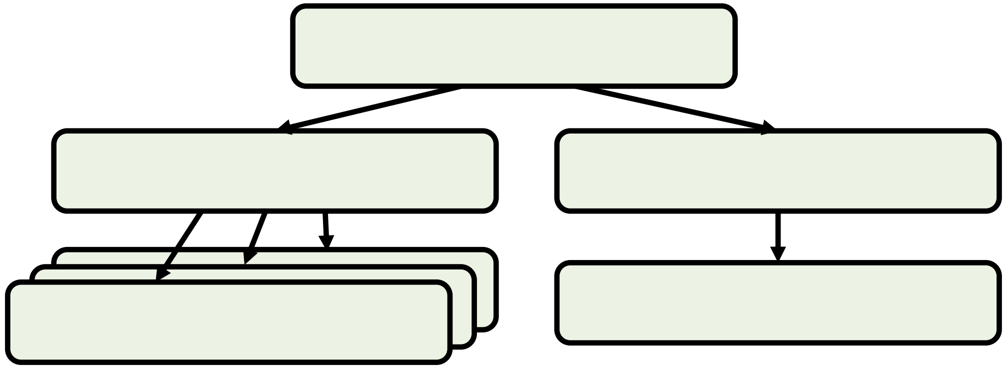


Original Radix Tree

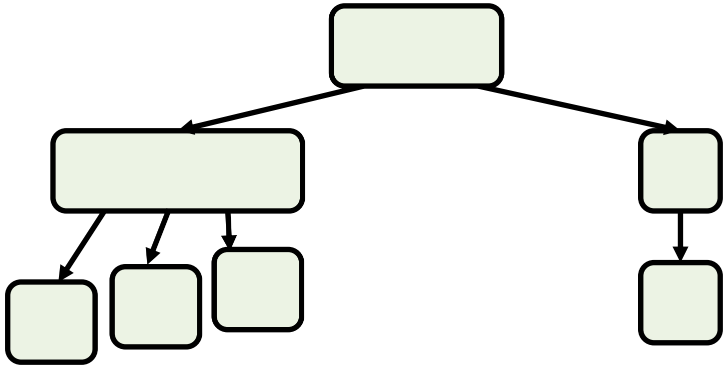


Optimization 1: adaptive node type

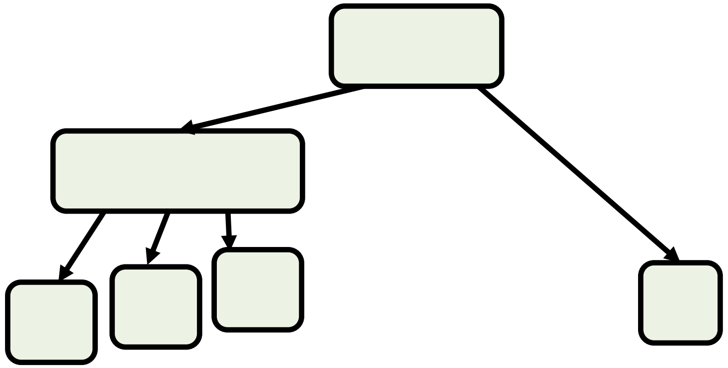
Key Idea: Adaptive Radix Tree



Original Radix Tree



Optimization 1: adaptive node type



Optimization 2: collapsing inner nodes

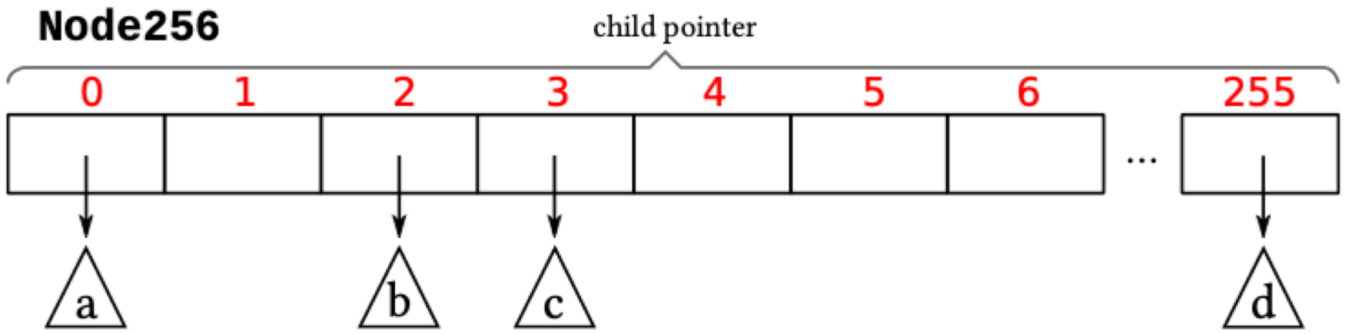
Inner Node Structure

Node4 and **Node16**

Node48

Node256

- 256 child pointers indexed with partial key byte directly
- (Same as original radix tree)



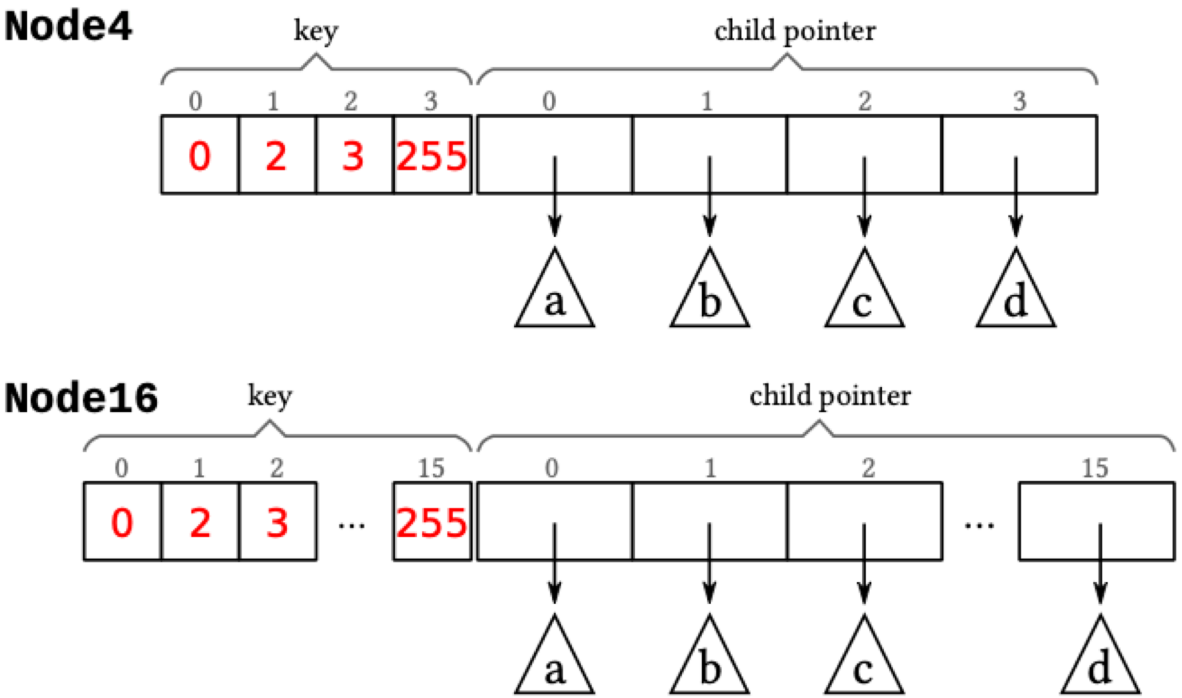
Inner Node Structure

Node4 and **Node16**

- Store up to 4 (16) partial keys and the corresponding pointers
- Each partial key is one byte

Node48

Node256

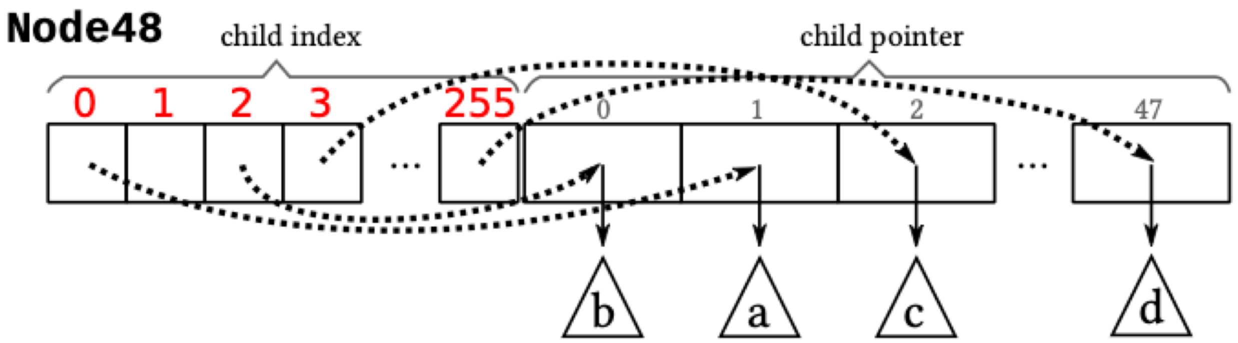


Inner Node Structure

Node4 and Node16

Node48

- 256 entries indexed with partial key byte directly
- Each entry stores a one-byte index to a child pointer array
- Child pointer array contains 48 pointers to children nodes



Node256

Collapsing Inner Node

Lazy expansion: remove path to single leaf

- Inner nodes created only required to distinguish at least two leaf nodes

Path compression: merge one-way node into child node

- Removes all inner nodes that have only a single child

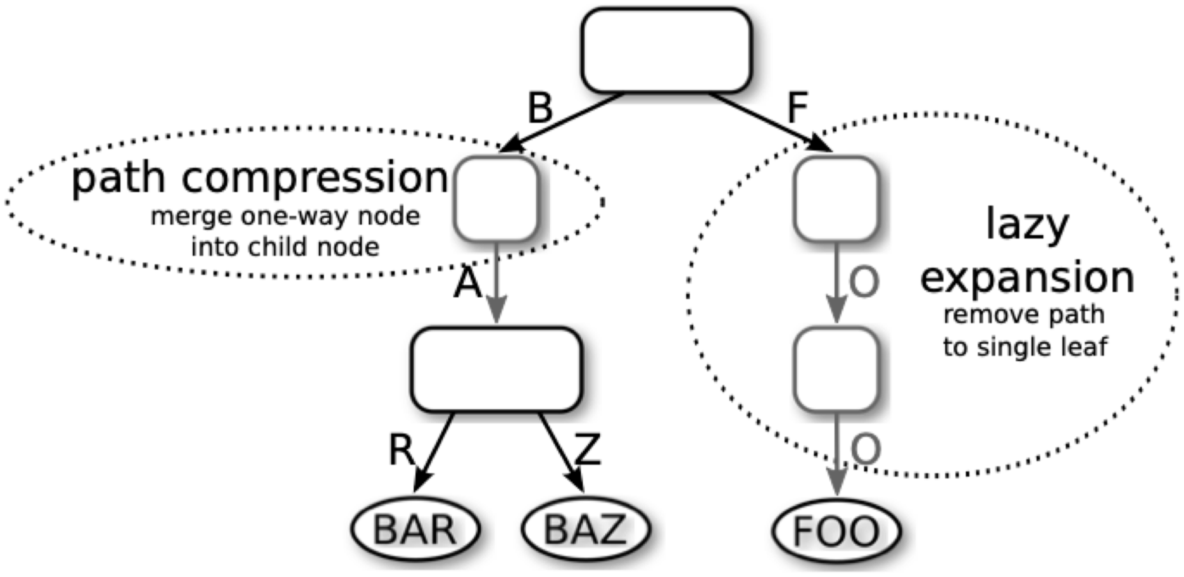


Fig. 6. Illustration of lazy expansion and path compression.

Collapsing Inner Node

Pessimistic

– Collapsed prefix key stored in each node as variable length key

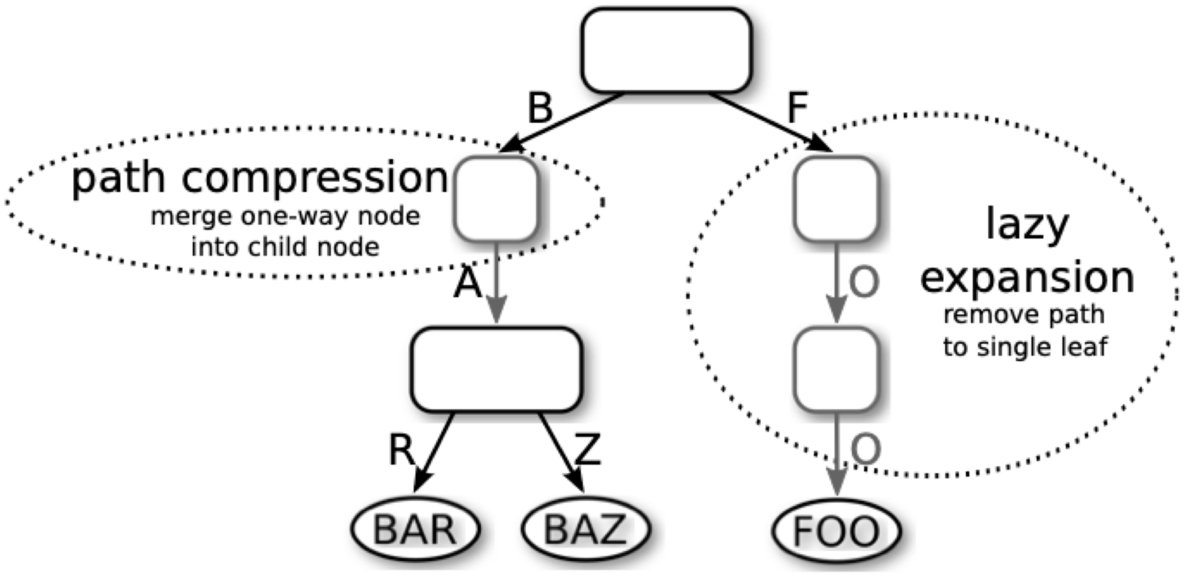


Fig. 6. Illustration of lazy expansion and path compression.

Collapsing Inner Node

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

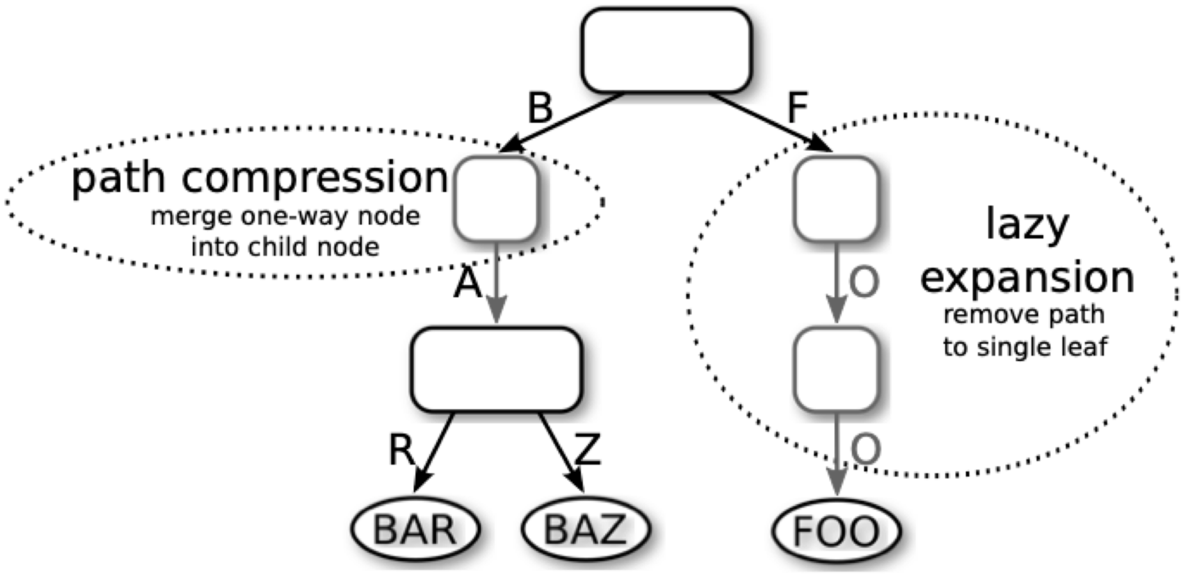


Fig. 6. Illustration of lazy expansion and path compression.

Collapsing Inner Node

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

- Store up to a constant-size collapsed key (8 bytes); once exceeded, switch to optimistic strategy

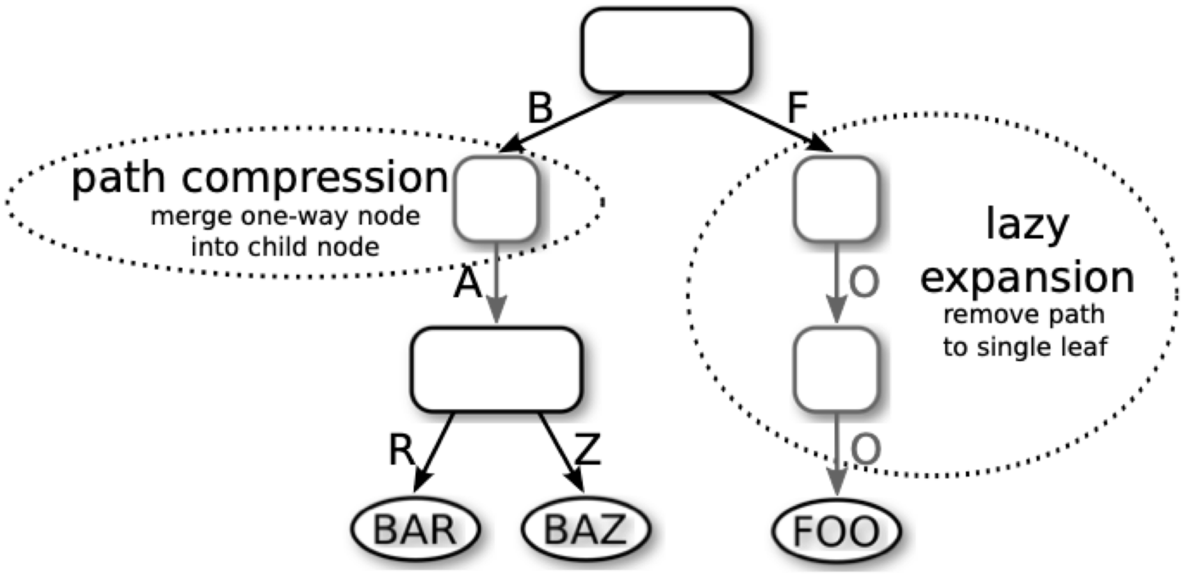
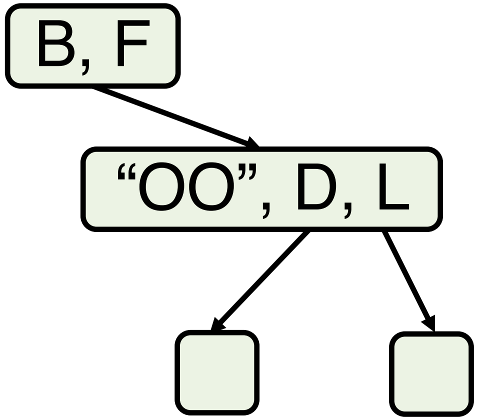


Fig. 6. Illustration of lazy expansion and path compression.

Search Algorithm

```
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)
```

Fig. 7. Search algorithm.



Example: search for **FOOD**

Insert Algorithm

```
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)
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)
```

Discussion

Space consumption

- ART requires at most 52 bytes of memory to index a key
- Q: What if the key itself is larger than 52 bytes?

Discussion

Space consumption

- ART requires at most 52 bytes of memory to index a key
- Q: What if the key itself is larger than 52 bytes?

Binary comparable keys

- For finite and totally ordered domains, always possible to transform values to binary-comparable keys

Evaluation—Single-Threaded Lookup

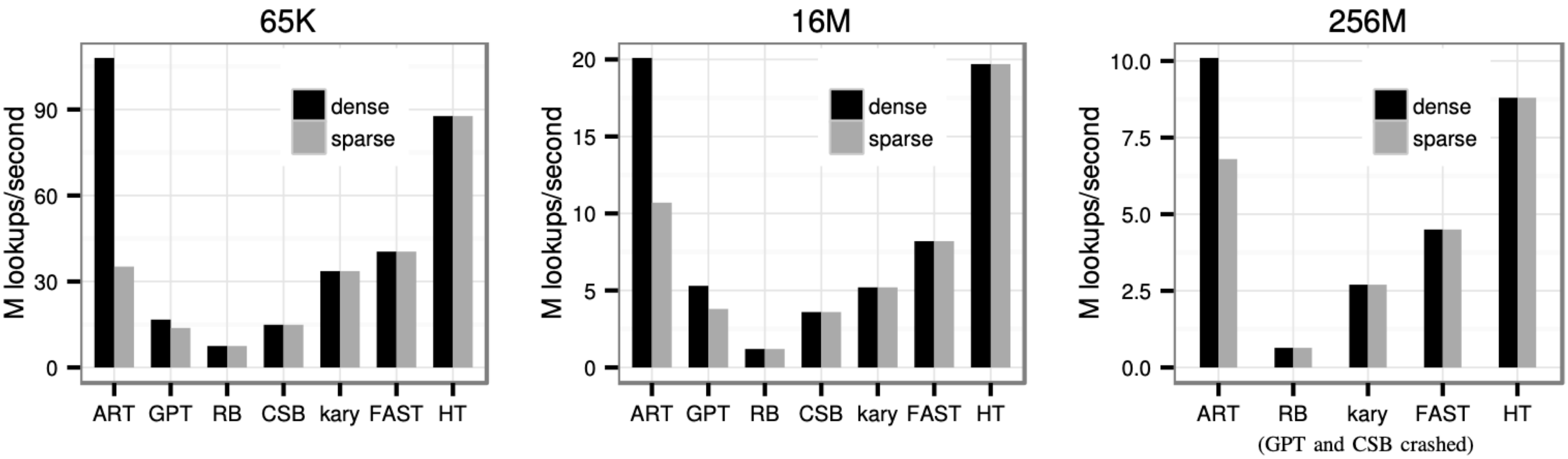


Fig. 10. Single-threaded lookup throughput in an index with 65K, 16M, and 256M keys.

Evaluation—Single-Threaded Insert

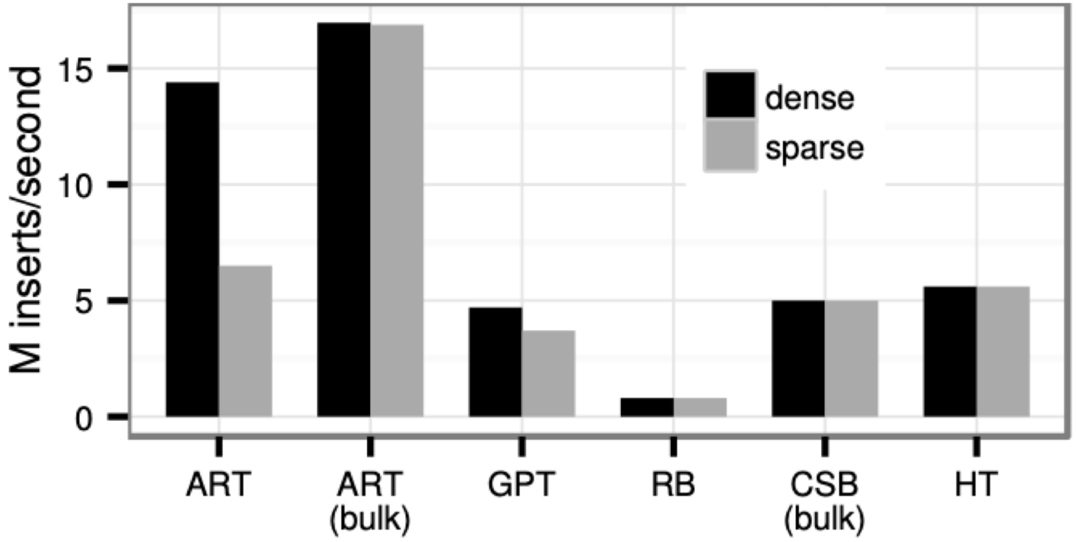


Fig. 14. Insertion of 16M keys into an empty index structure.

Evaluation—Single-Threaded Insert

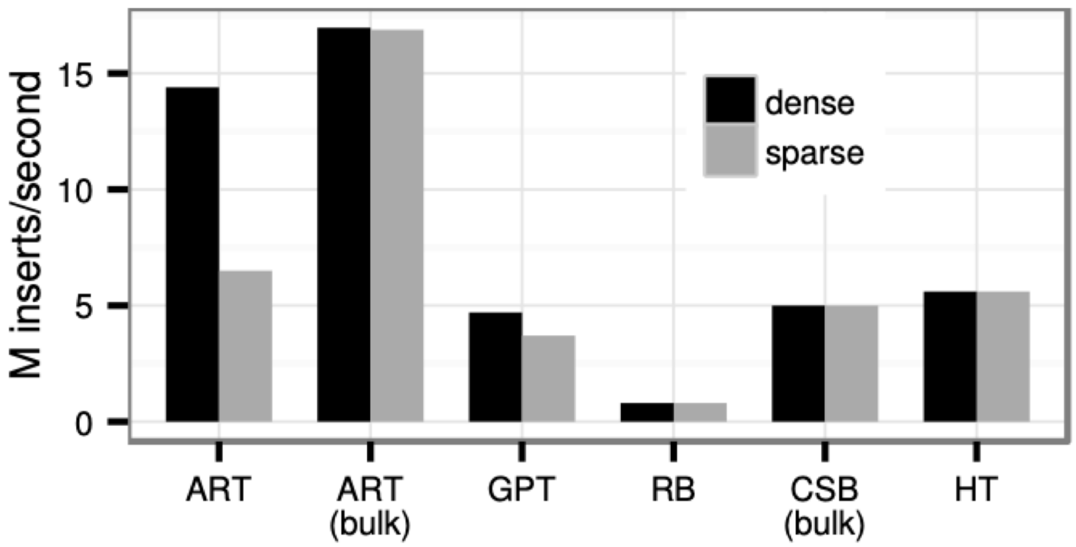


Fig. 14. Insertion of 16M keys into an empty index structure.

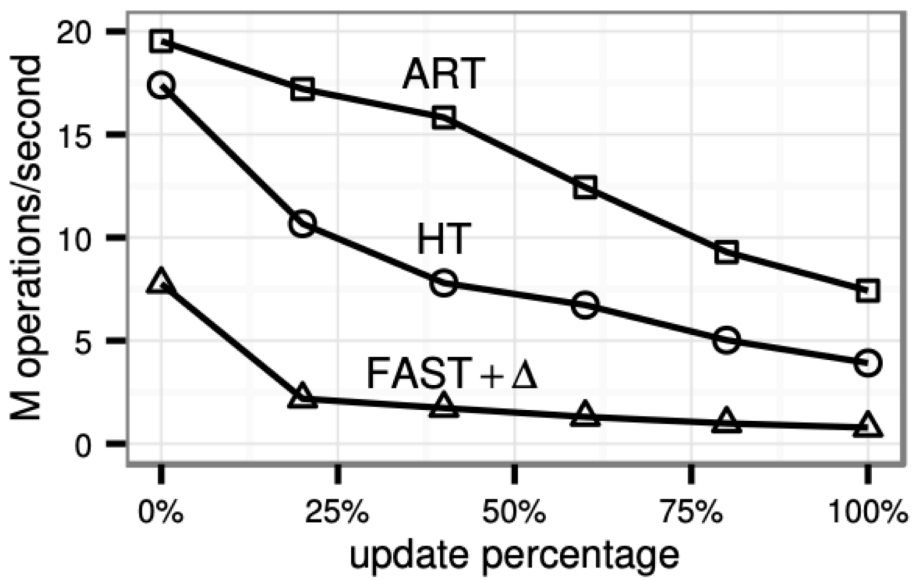


Fig. 15. Mix of lookups, insertions, and deletions (16M keys).

Evaluation – More Baselines

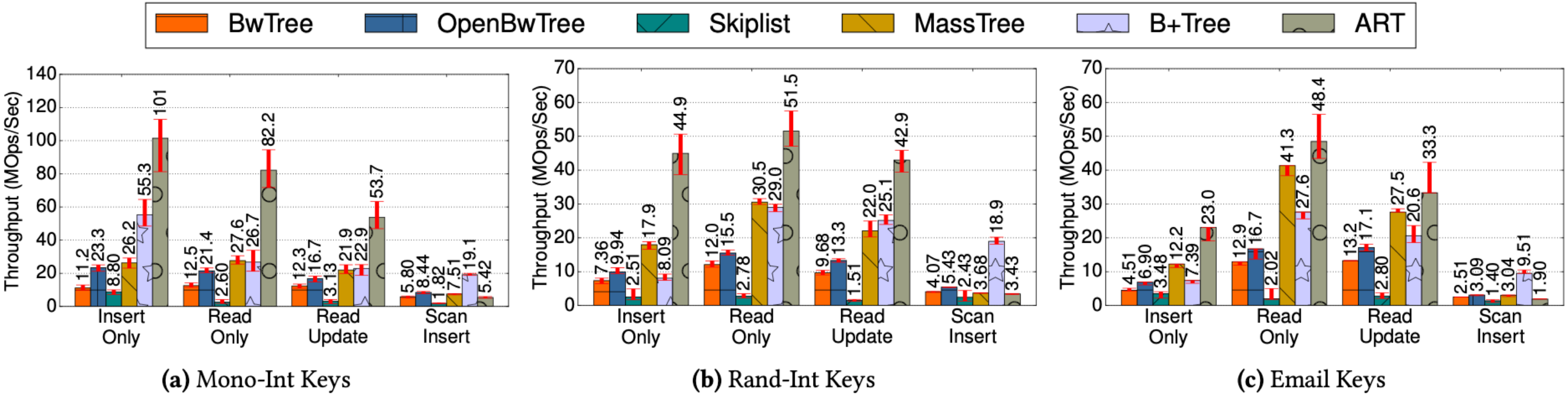
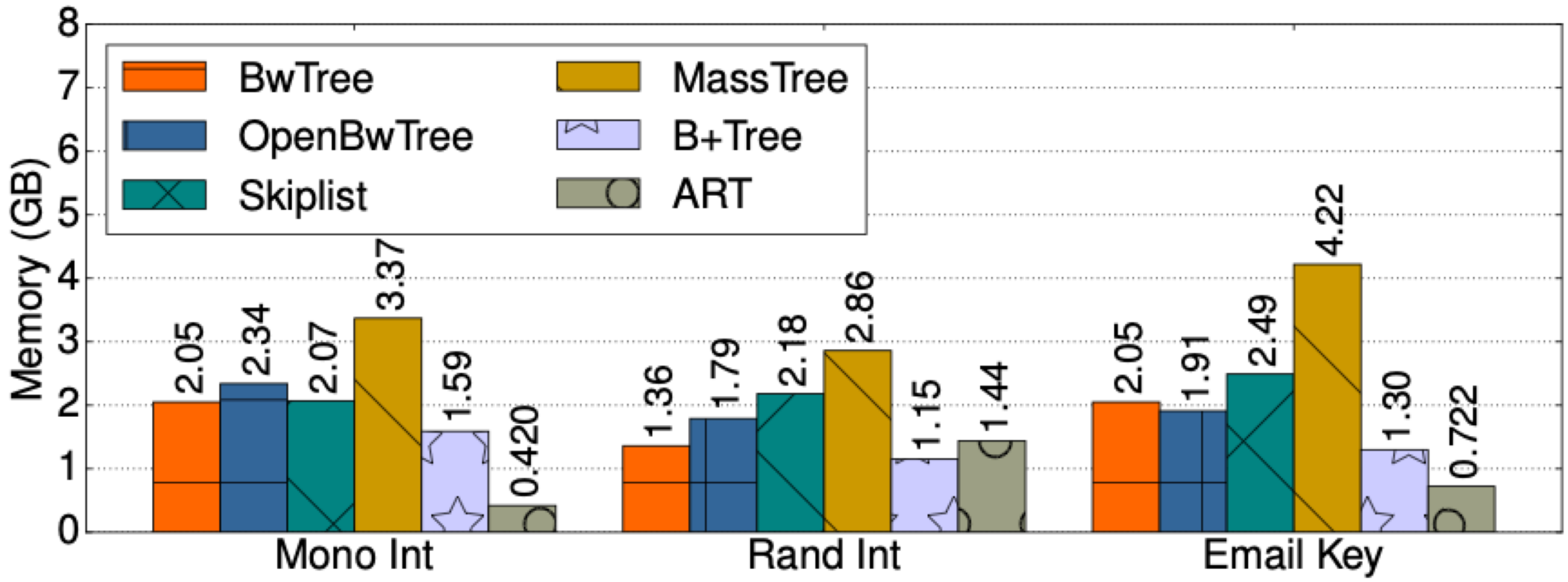


Figure 14: In-Memory Index Comparison (Multi-Threaded) – 20 worker threads. All worker threads are pinned to NUMA node 0.

* Wang, Ziqi, et al. *Building a bw-tree takes more than just buzz words*. SIGMOD 2018

Evaluation – Memory Usage



(b) Multi-Threaded – Read/Update

* Wang, Ziqi, et al. *Building a bw-tree takes more than just buzz words*. SIGMOD 2018

Q/A – Adaptive Radix Tree

Use of SIMD in realistic DBs?

Can ART fit well in distributed systems?

Concurrent operations in ART?

Keys that are prefixes of other keys?

ART vs. B-link tree?

What if data does not fit in memory?

Next Week

Philip Bernstein, et al., Concurrency Control and Recovery in Database Systems, Chapter 6. Addison-wesley, 1987

- Skip 6.4 (covered next lecture), 6.5, 6.7, and exercise
- About 20 pages to read

C. Mohan, et al. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. ACM Transactions on Database Systems, 1992

- Skip Section 1 and everything after (including) Section 8
- May also skip Section 2
- About 25–30 pages to read