CS 764: Topics in Database Management Systems

Lecture 16: Adaptive Radix Tree

Xiangyao Yu

10/31/2022

Announcement

Midterm exam

- All papers before the exam are included
- Guest lectures are **not** included
- **Nov. 9 (Wednesday) noon–Nov. 11 (Friday) noon**, central time
- The exam questions will be posted on Piazza (as a word document)
- Please use Piazza to ask questions privately
- Email your answers to the TA by the deadline

Suggested ways to submit your solutions:

- Directly type your solutions in this MS word document
- Print out the exam and submit a photocopy of your solutions
- Convert the exam into a pdf file and write your solutions on it

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

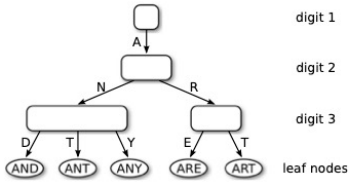


Fig. 1. Adaptively sized nodes in our radix tree.

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

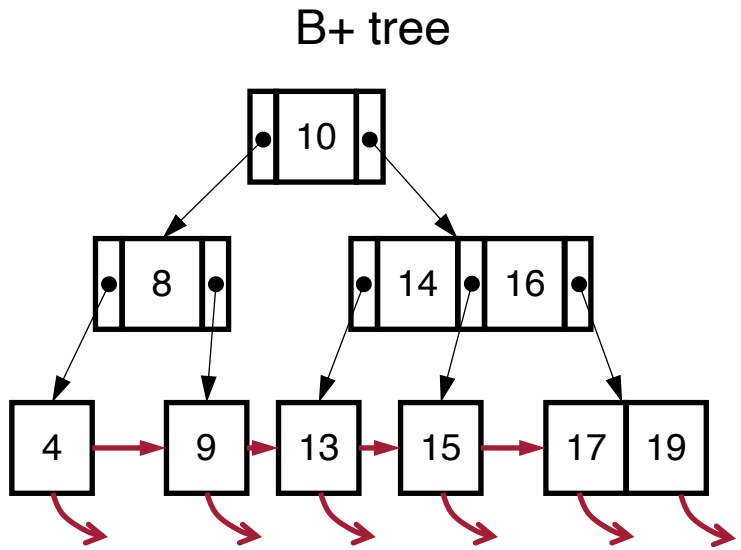
Evaluation

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)

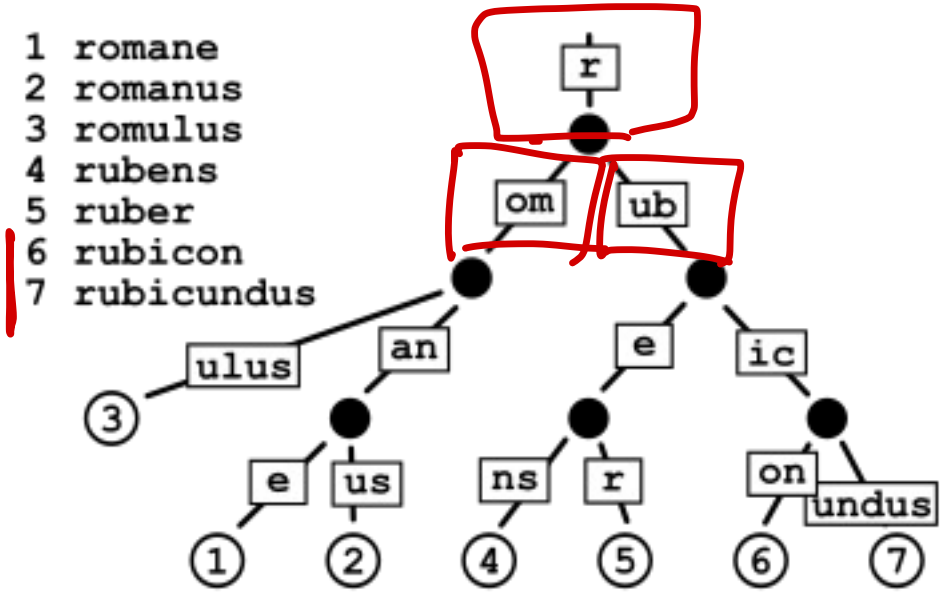


Trie (aka. digital tree or prefix tree)

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

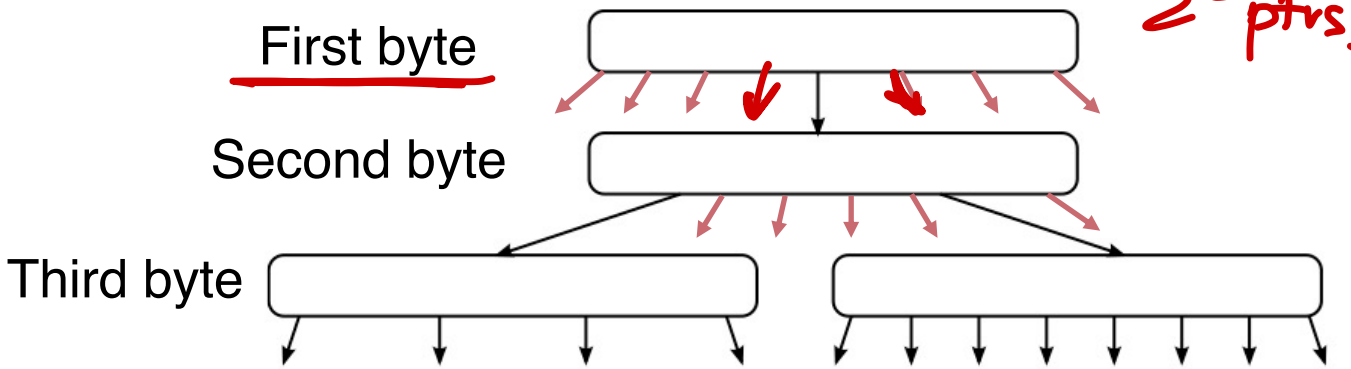


Source: https://en.wikipedia.org/wiki/Radix_tree

Static Radix Tree

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

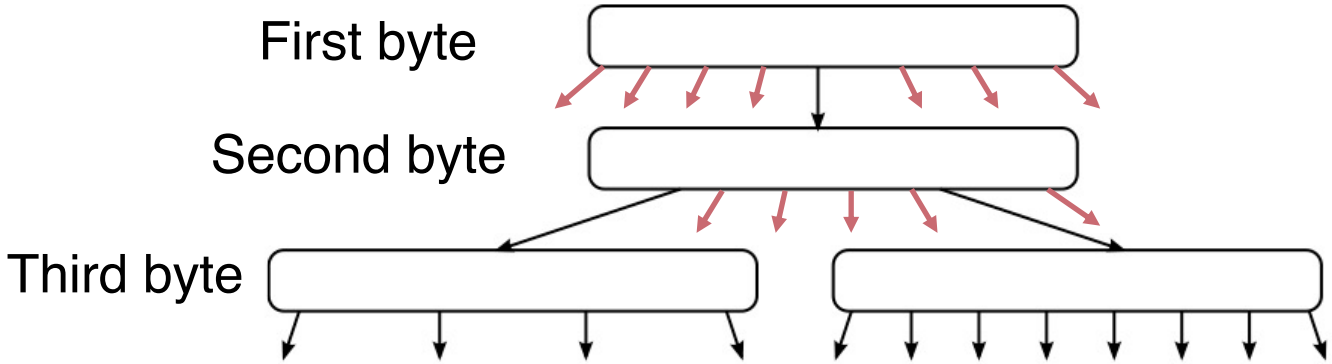
An inner node is an array of 2^s pointers



Static Radix Tree

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

An inner node is an array of 2^s pointers



Example:

k = 32 bit keys

Consider the index space consumption to insert one key

Extra space ≈ $\left\lceil \frac{k}{s} \right\rceil$ (# of levels) × 2^s (node size per level)

Static Radix Tree

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

An inner node is an array of 2^s pointers

$\lceil \frac{k}{s} \rceil$

Large span

=> reduced height

=> exponential tree size

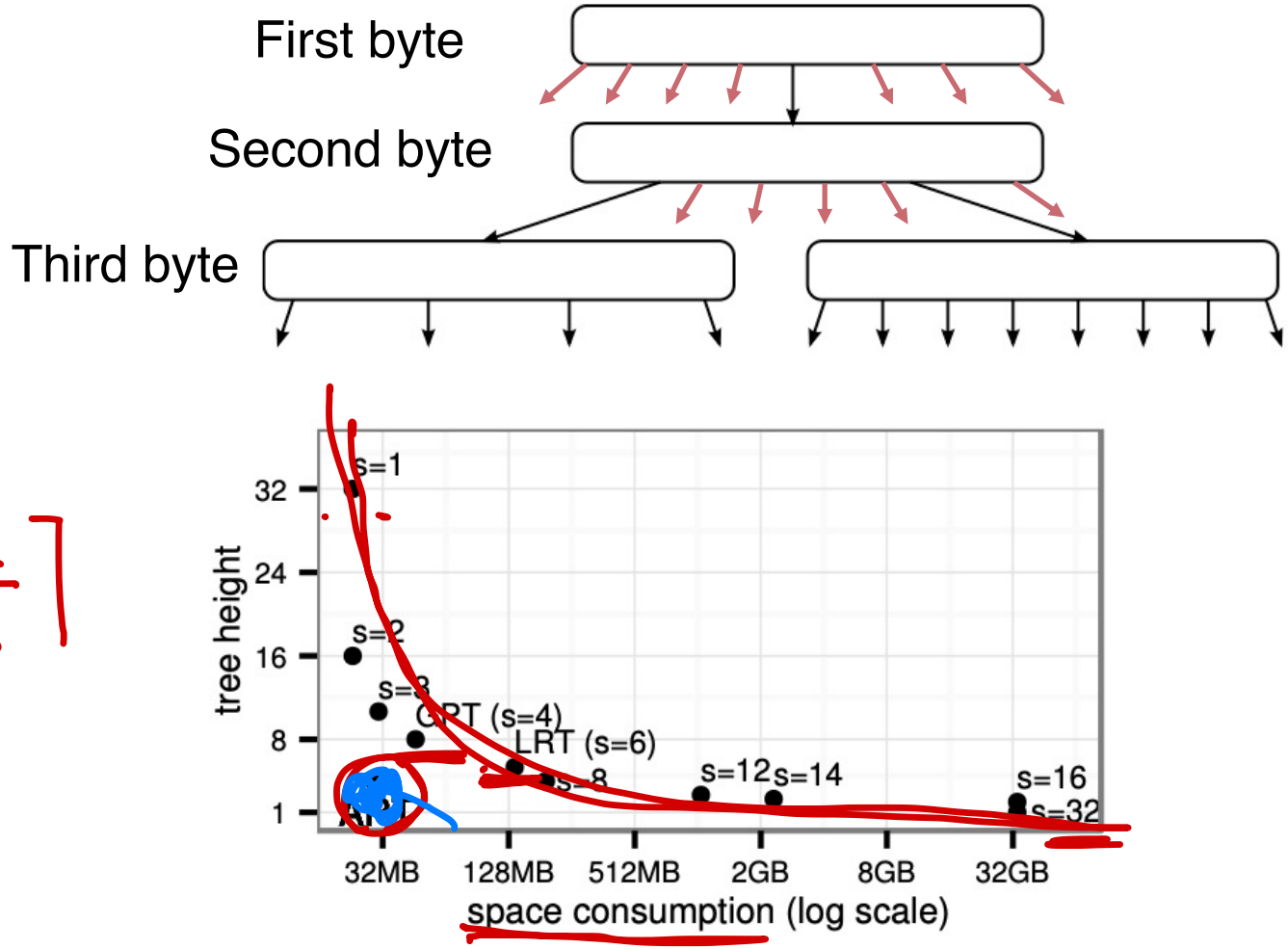
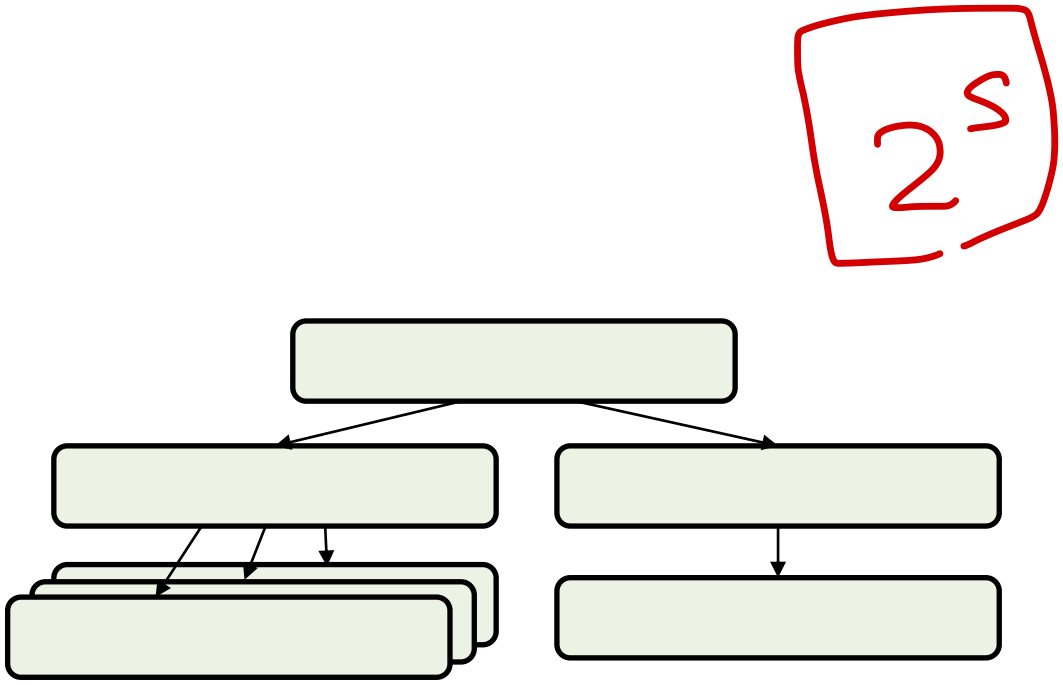


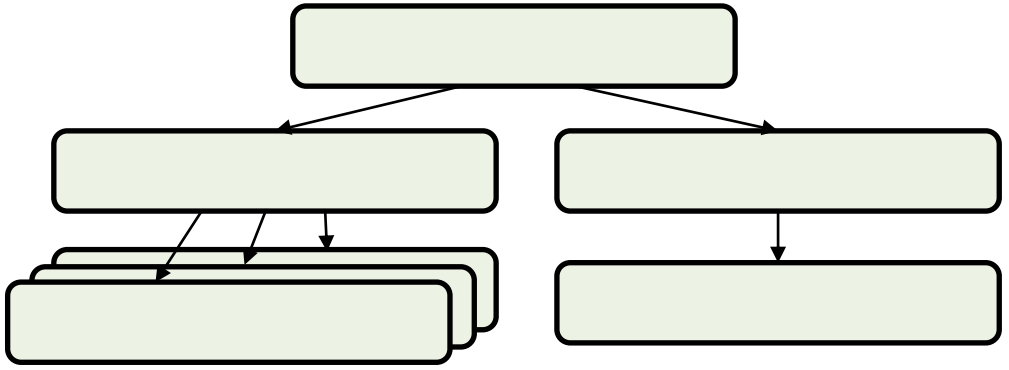
Fig. 3. Tree height and space consumption for different values of the span parameter s when storing 1M uniformly distributed 32 bit integers. Pointers are 8 byte long and nodes are expanded lazily.

Key Idea: Adaptive Radix Tree

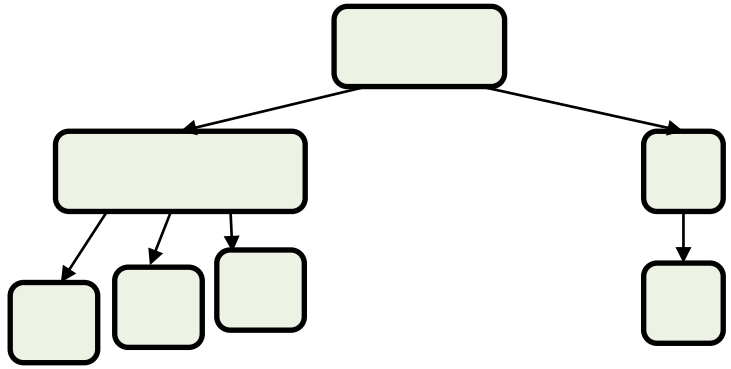


Original Radix Tree

Key Idea: Adaptive Radix Tree



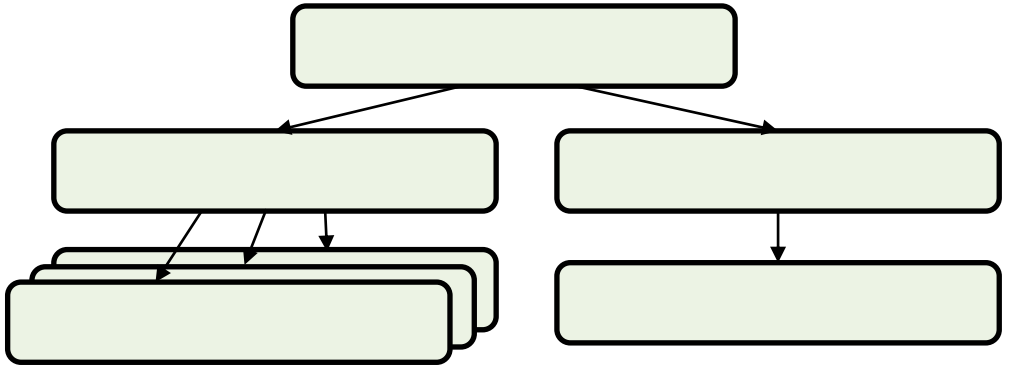
Original Radix Tree



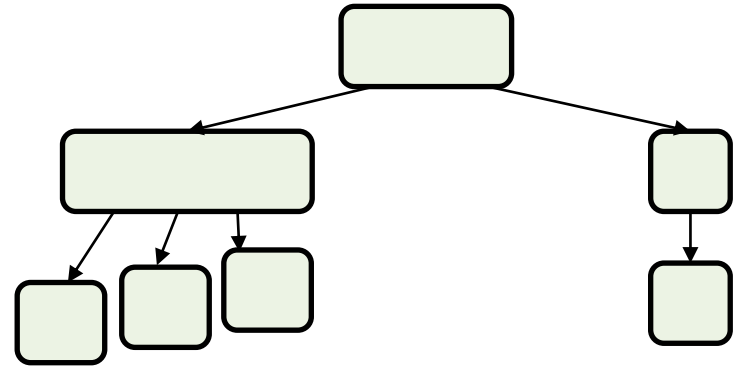
Optimization 1: adaptive node type

Key idea: Use a small node type when only a small number of children pointers exist

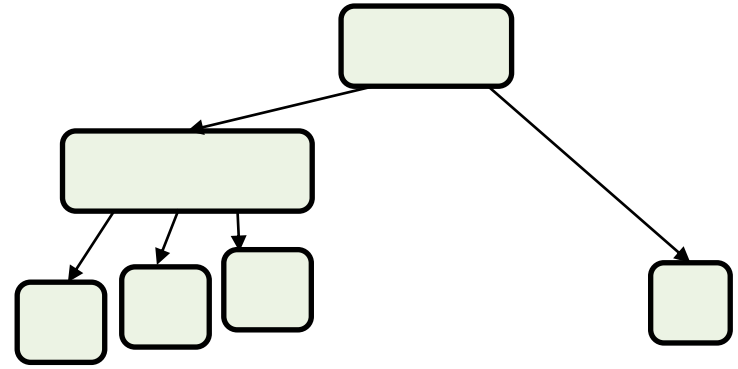
Key Idea: Adaptive Radix Tree



Original Radix Tree



Optimization 1: adaptive node type



Optimization 2: collapsing inner nodes

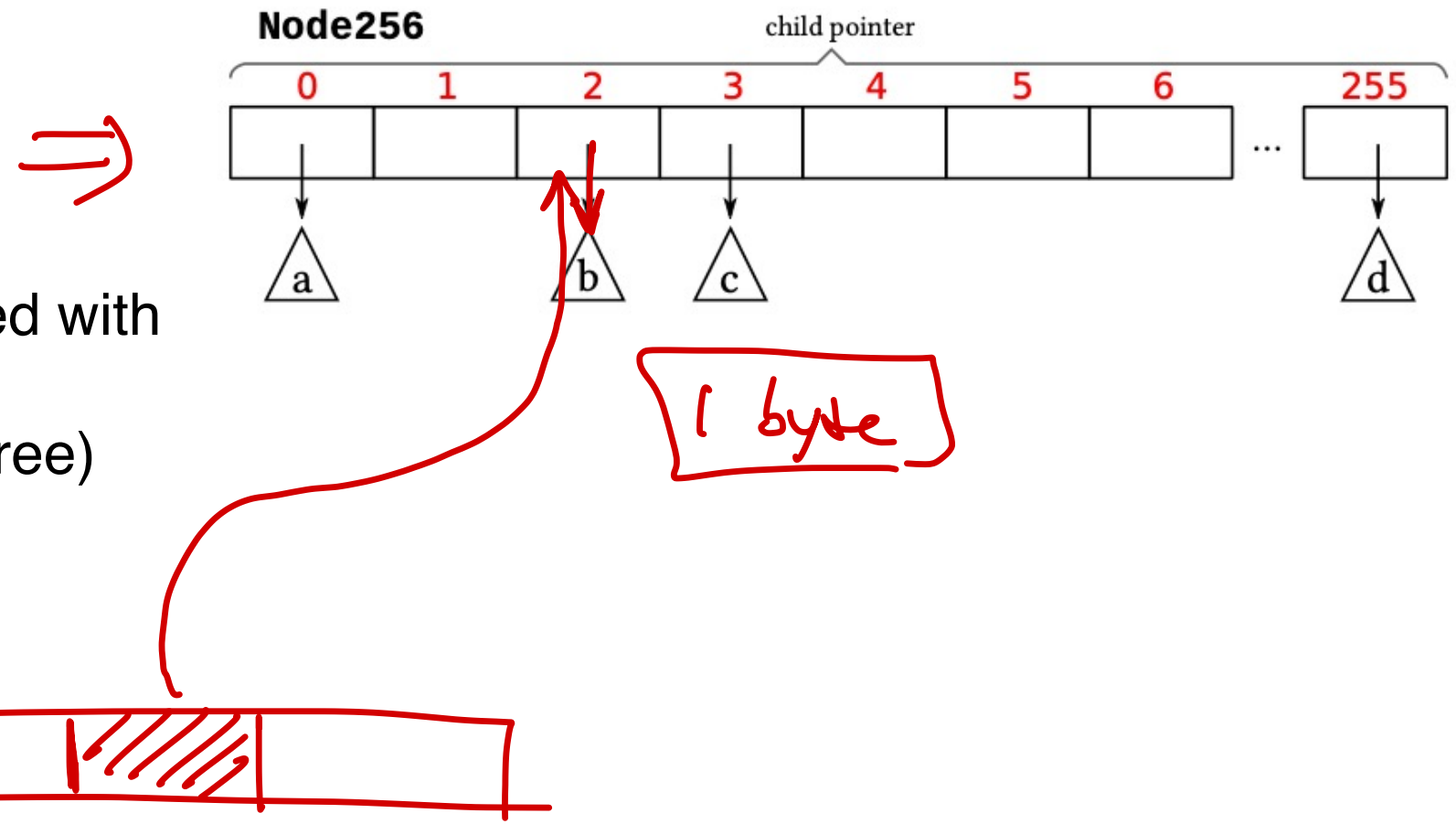
Inner Node Structure

Node4 and **Node16**

Node48

Node256

- 256 child pointers indexed with partial key byte directly
- (Same as original radix tree)
- Used for 49–256 entries



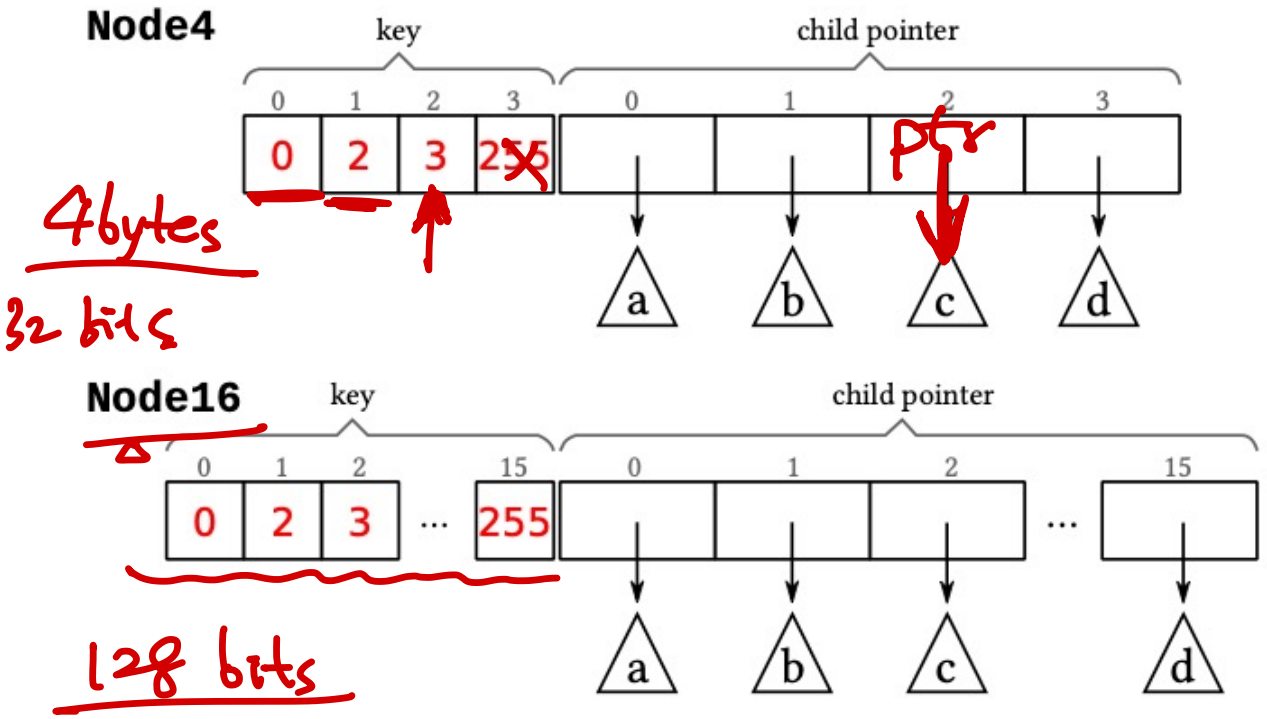
Inner Node Structure

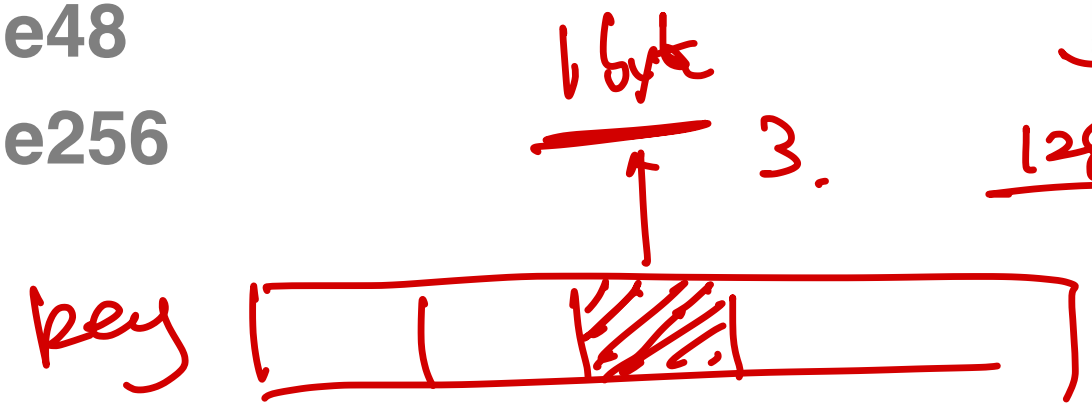
Node4 and **Node16**

- Store up to 4 (16) partial keys and the corresponding pointers
- Each partial key is one byte
- Use SIMD instructions to accelerate key search

Node48

Node256





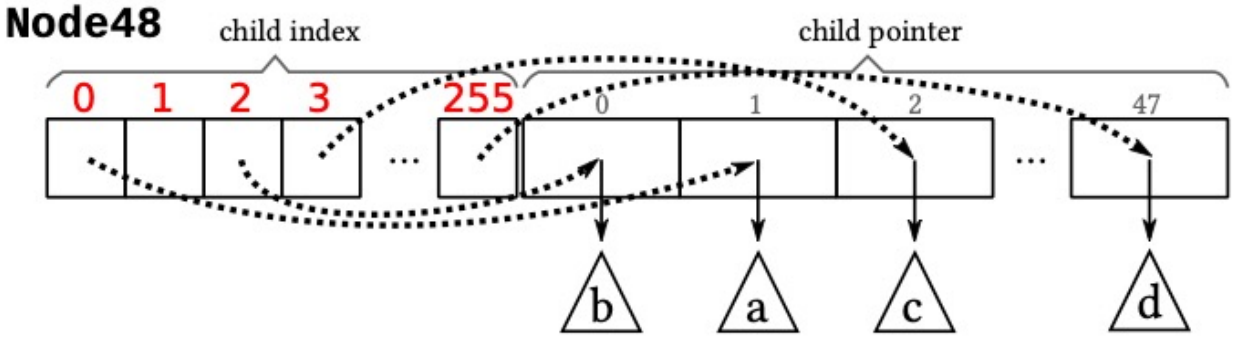
Inner Node Structure

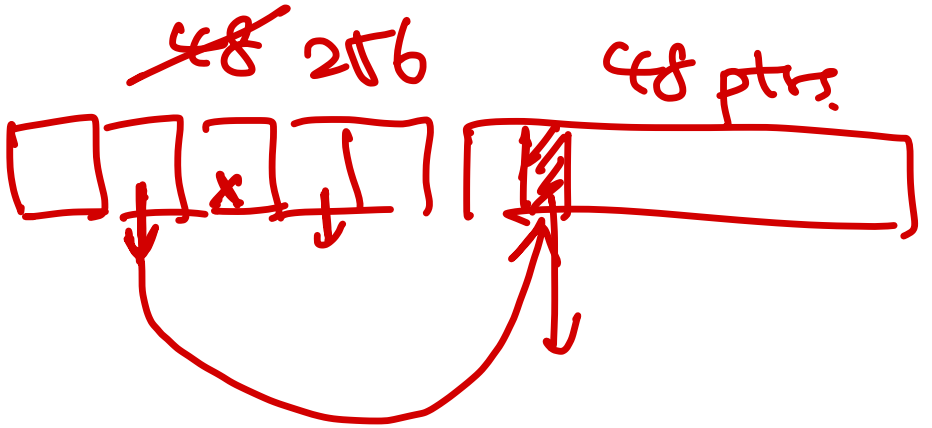
Node4 and **Node16**

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





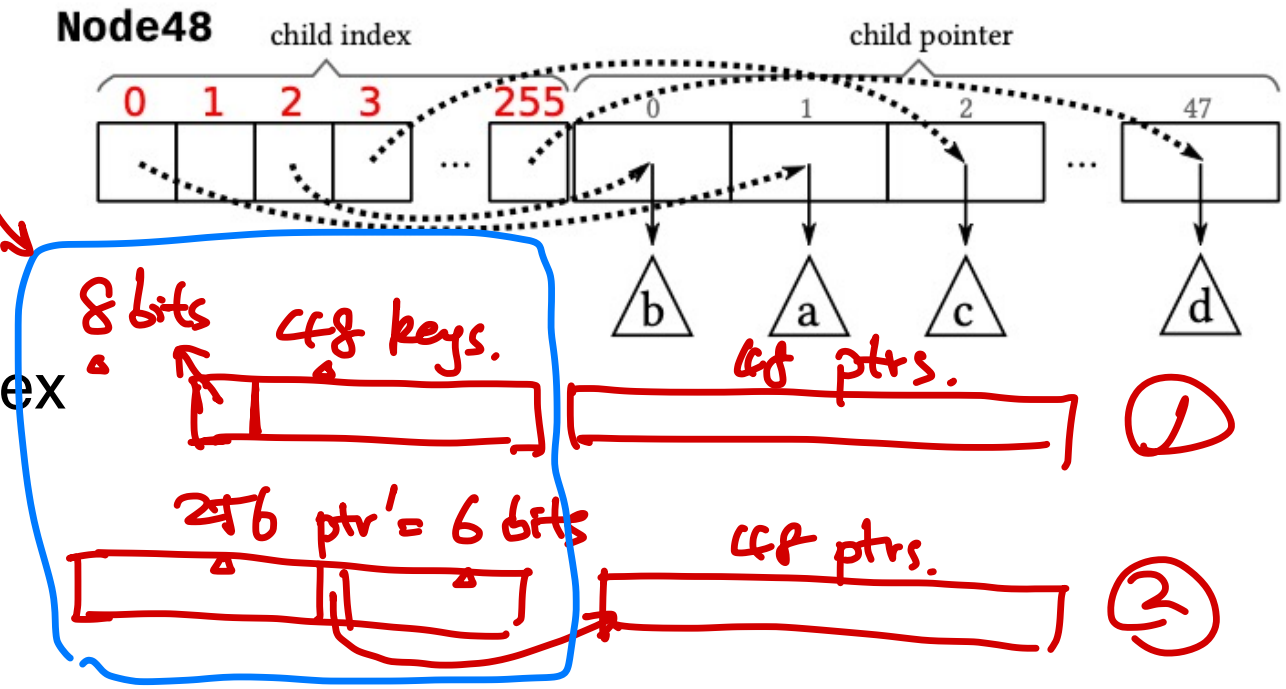
Inner Node Structure

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



Discussion Question

Q1: Is Node48 more space efficient compared to Node4/16 layout?

Q2: What is the key advantage of Node48 layout?

Collapsing Inner Node

FAR

Lazy expansion: remove path to single leaf

- Inner nodes created only required to distinguish at least two leaf nodes
- In the example, root can directly point to leaf FOO, eliminating the two inner nodes
- Requires the key to be stored at the leaf or in the database

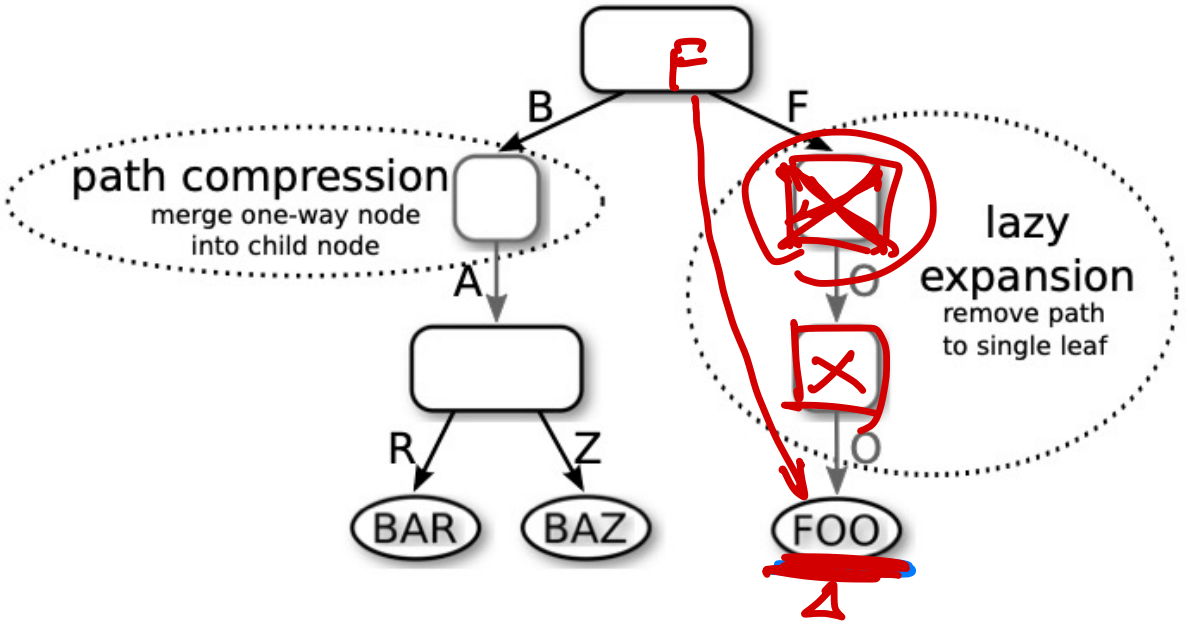


Fig. 6. Illustration of lazy expansion and path compression.

Collapsing Inner Node

Lazy expansion: remove path to single leaf

Path compression: merge one-way node into child node

- Removes all inner nodes that have only a single child

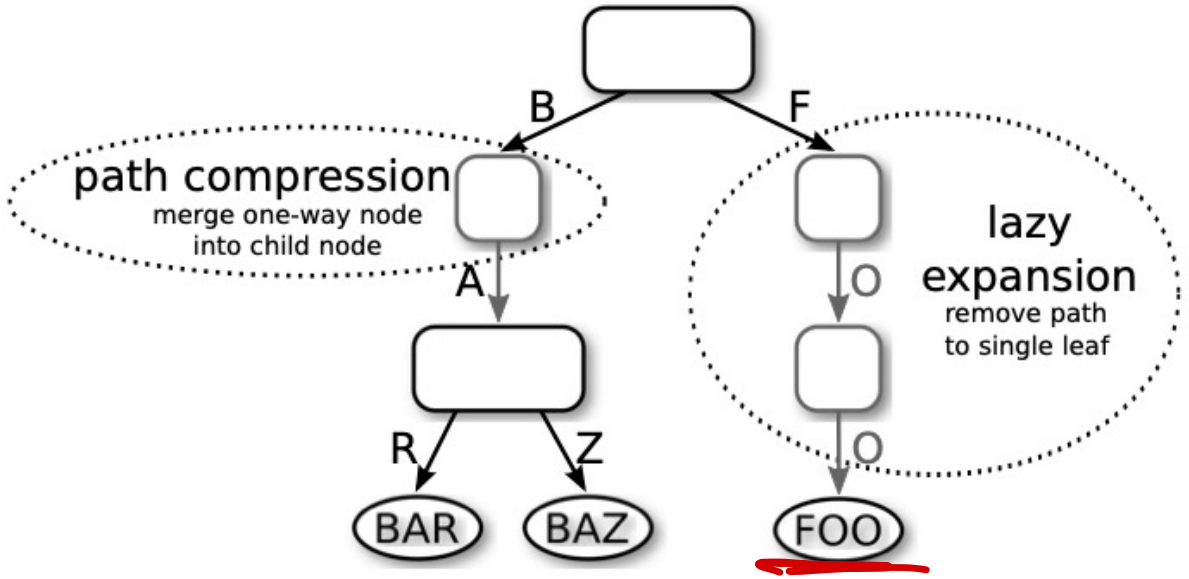


Fig. 6. Illustration of lazy expansion and path compression.

Collapsing Inner Node

Lazy expansion: remove path to single leaf

Path compression: merge one-way node into child node

- Removes all inner nodes that have only a single child
- **Pessimistic**: child node stores the compressed partial key

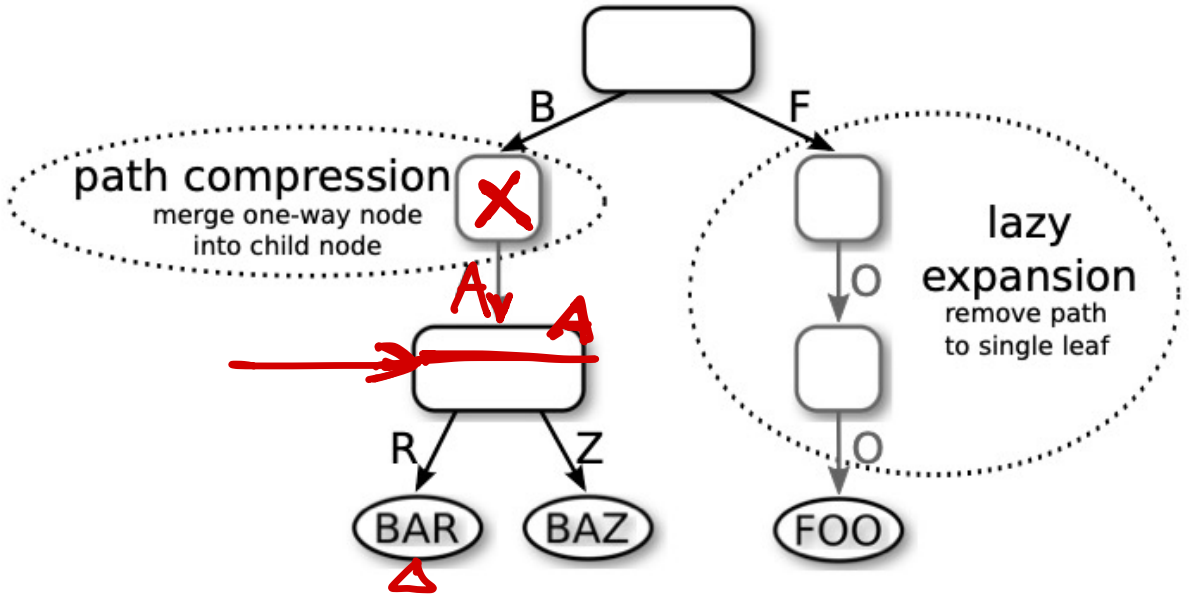


Fig. 6. Illustration of lazy expansion and path compression.

Collapsing Inner Node

Lazy expansion: remove path to single leaf

Path compression: merge one-way node into child node

- Removes all inner nodes that have only a single child
- **Pessimistic**: child node stores the compressed partial key
- **Optimistic**: child node stores only the *length* of compressed partial key

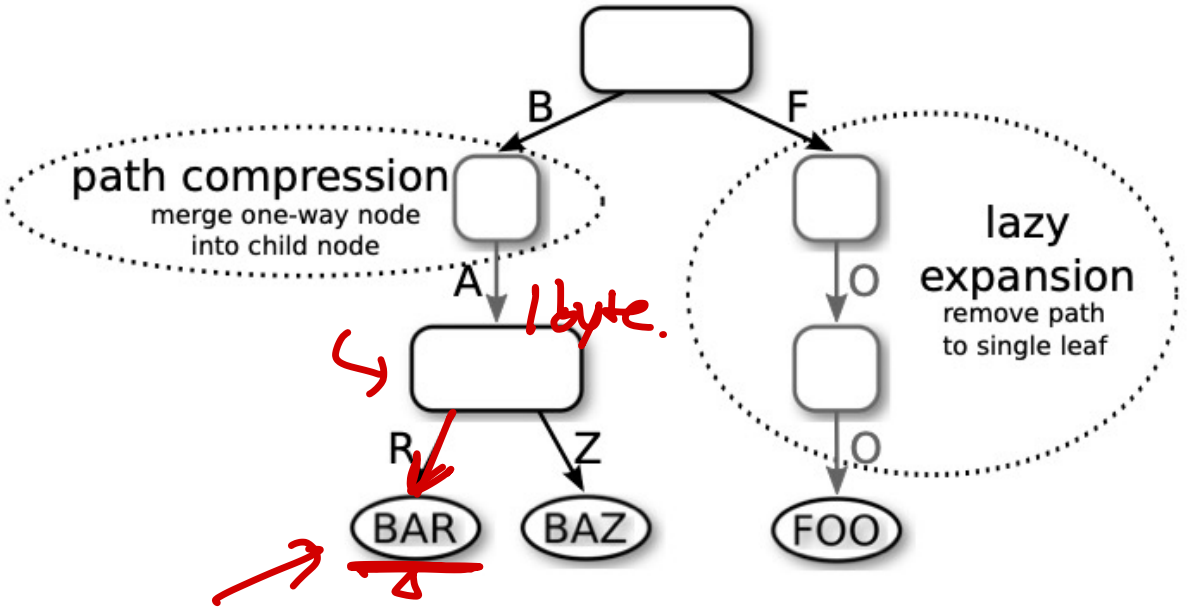


Fig. 6. Illustration of lazy expansion and path compression.

Collapsing Inner Node

Lazy expansion: remove path to single leaf

Path compression: merge one-way node into child node

- Removes all inner nodes that have only a single child
- **Pessimistic**: child node stores the compressed partial key
- **Optimistic**: child node stores only the *length* of compressed partial key
- **Hybrid**: use constant vector to store partial key, switch to optimistic approach if the vector overflows

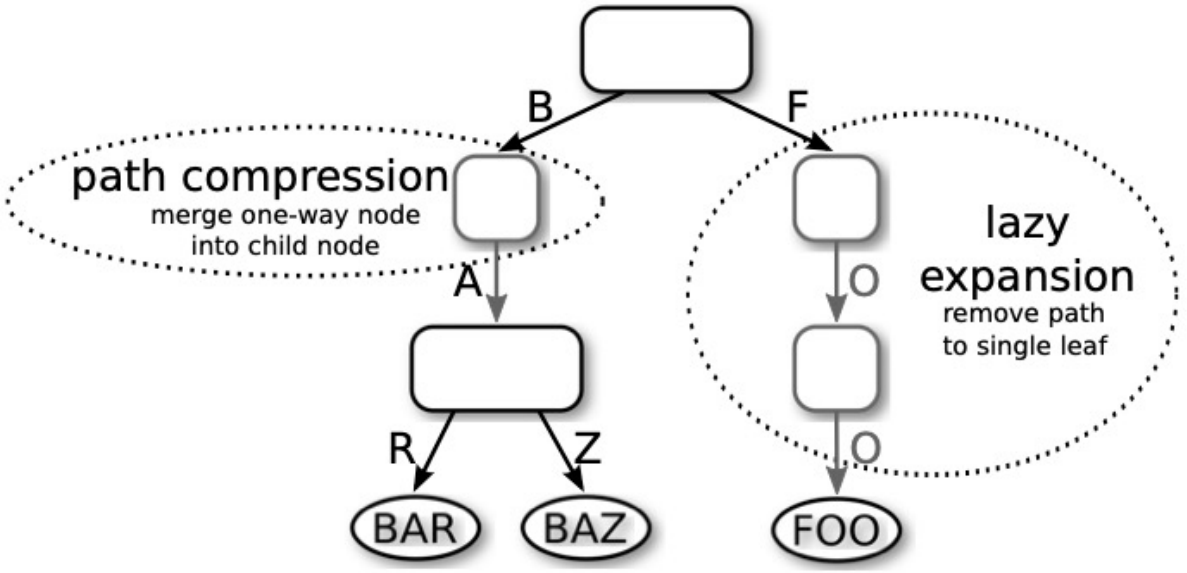
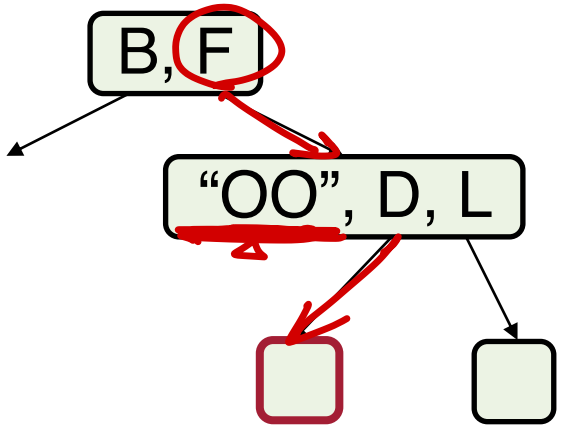


Fig. 6. Illustration of lazy expansion and path compression.

Search Algorithm

```
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.



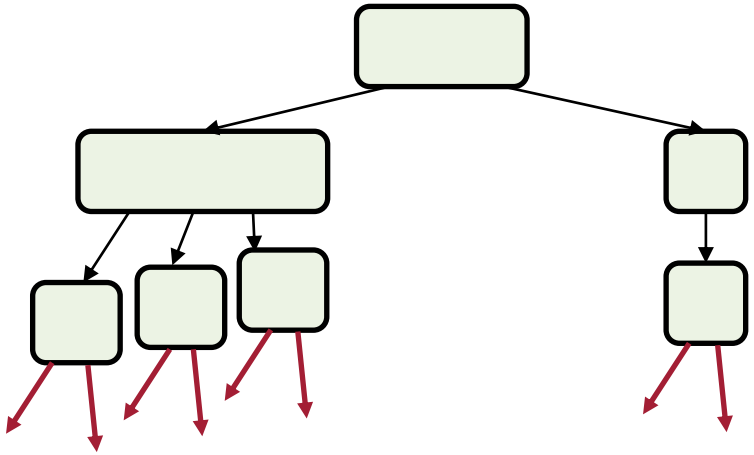
Example: search for **FOOD**

FEED

Space Consumption

ART requires at most **52 bytes** of memory to index a key

- Assume each child node has 52 byte budget, show that each node will have at least 52 bytes budget remain

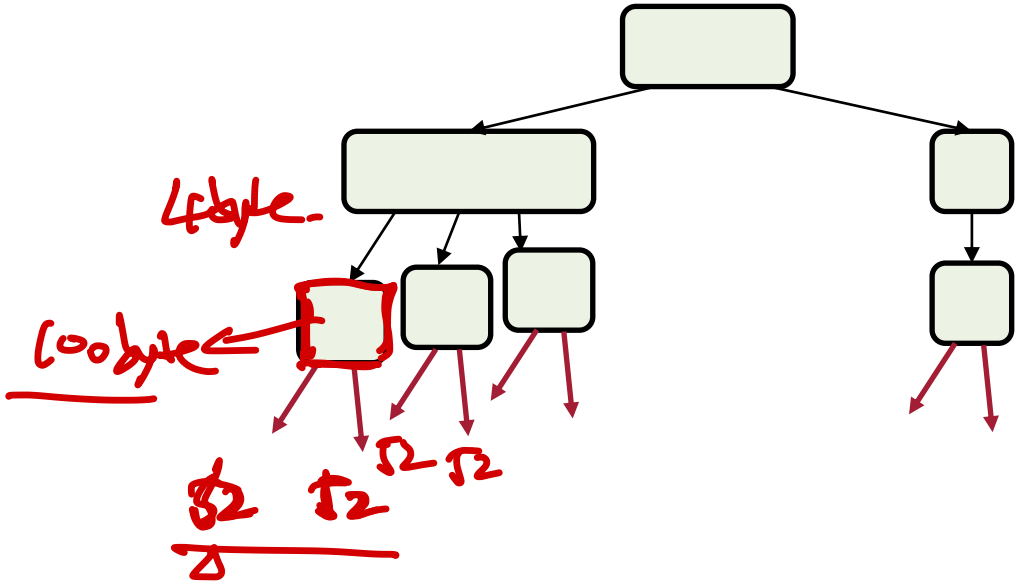


Space Consumption

ART requires at most **52 bytes** of memory to index a key

– Assume each child node has 52 byte budget, show that each node will have at least 52 bytes budget remain

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

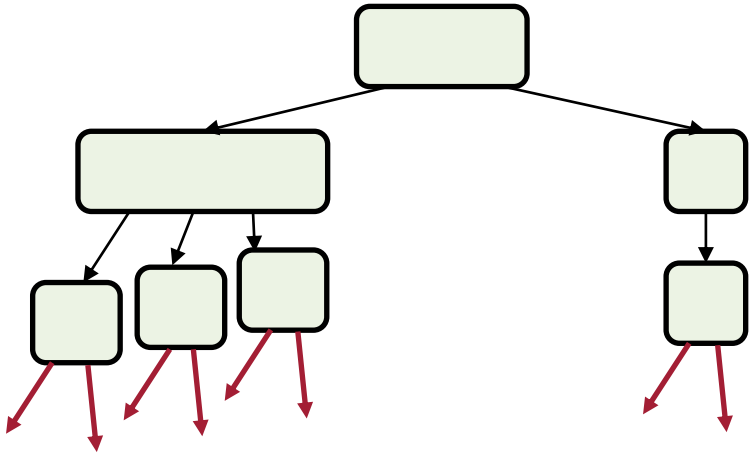


Space Consumption

ART requires at most **52 bytes** of memory to index a key

– Assume each child node has 52 byte budget, show that each node will have at least 52 bytes budget remain

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



Node4: b(n) ≤ 52 * 2 – 52 = **52**

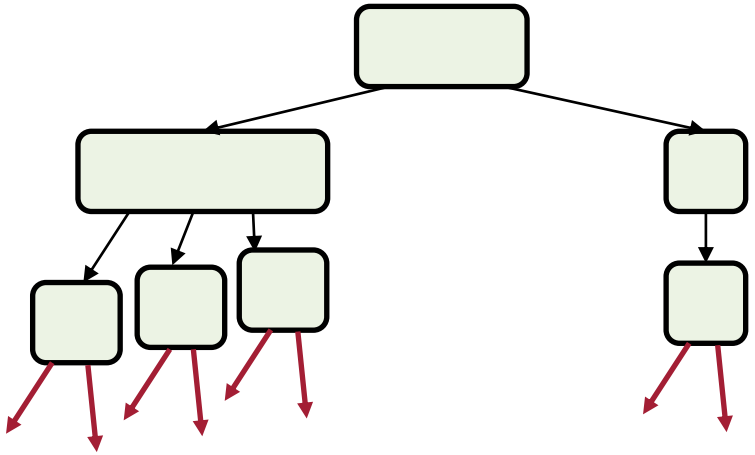
Type	Children	Space (bytes)
Node4	2-4	$16 + 4 + 4 \cdot 8 = 52$
Node16	5-16	$16 + 16 + 16 \cdot 8 = 160$
Node48	17-48	$16 + 256 + 48 \cdot 8 = 656$
Node256	49-256	$16 + 256 \cdot 8 = 2064$

Space Consumption

ART requires at most **52 bytes** of memory to index a key

– Assume each child node has 52 byte budget, show that each node will have at least 52 bytes budget remain

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



Node4: b(n) ≤ 52 * 2 – 52 = **52**

Node16: b(n) ≤ 52 * 5 – 160 = 100

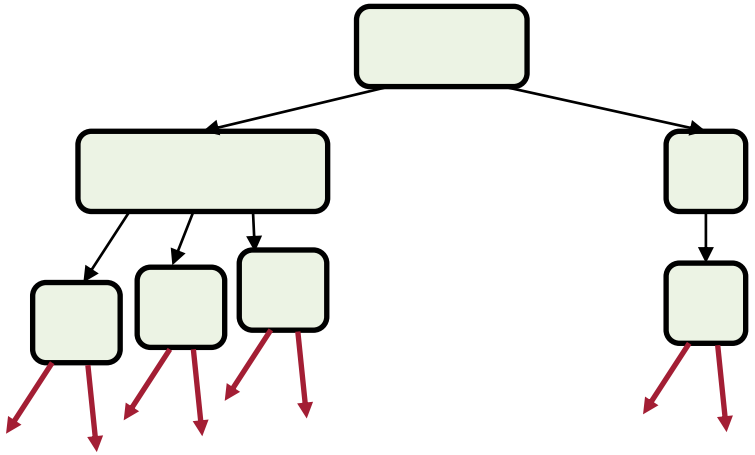
Type	Children	Space (bytes)
Node4	2-4	$16 + 4 + 4 \cdot 8 = 52$
Node16	5-16	$16 + 16 + 16 \cdot 8 = 160$
Node48	17-48	$16 + 256 + 48 \cdot 8 = 656$
Node256	49-256	$16 + 256 \cdot 8 = 2064$

Space Consumption

ART requires at most **52 bytes** of memory to index a key

– Assume each child node has 52 byte budget, show that each node will have at least 52 bytes budget remain

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



Node4: b(n) ≤ 52 * 2 – 52 = **52**
Node16: b(n) ≤ 52 * 5 – 160 = 100
Node48: b(n) ≤ 52 * 17 – 656 = 228
Node256: b(n) ≤ 52 * 49 – 2064 = 484

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	$16 + 256 \cdot 8 = 2064$

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

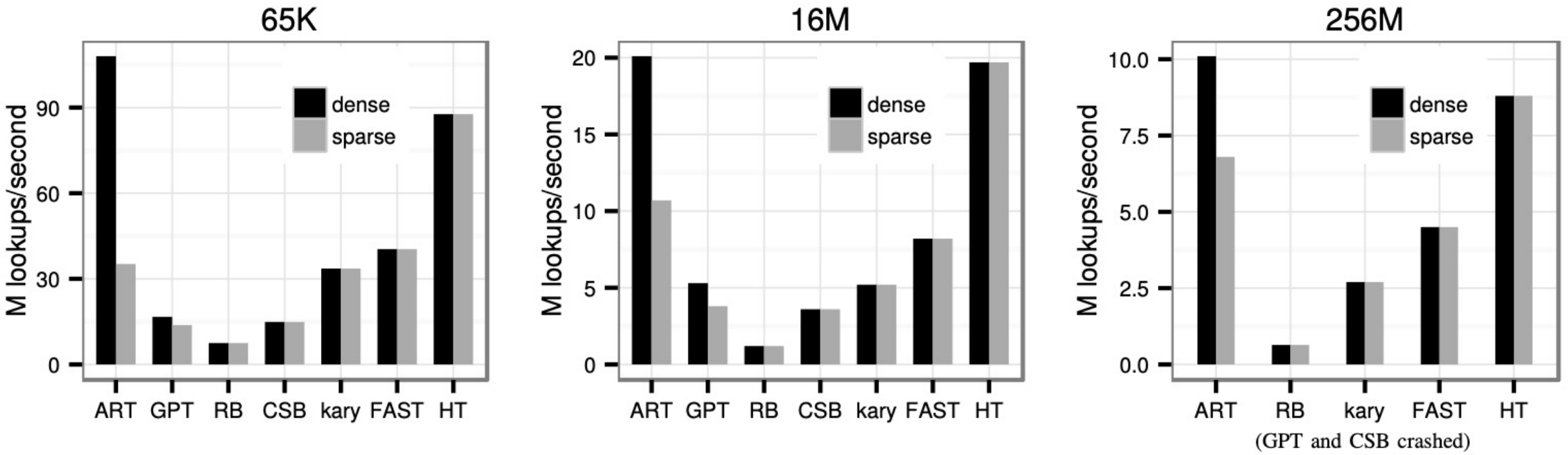


Fig. 10. Single-threaded lookup throughput in an index with 65K, 16M, and 256M keys.

Evaluation—Single-Threaded Insert

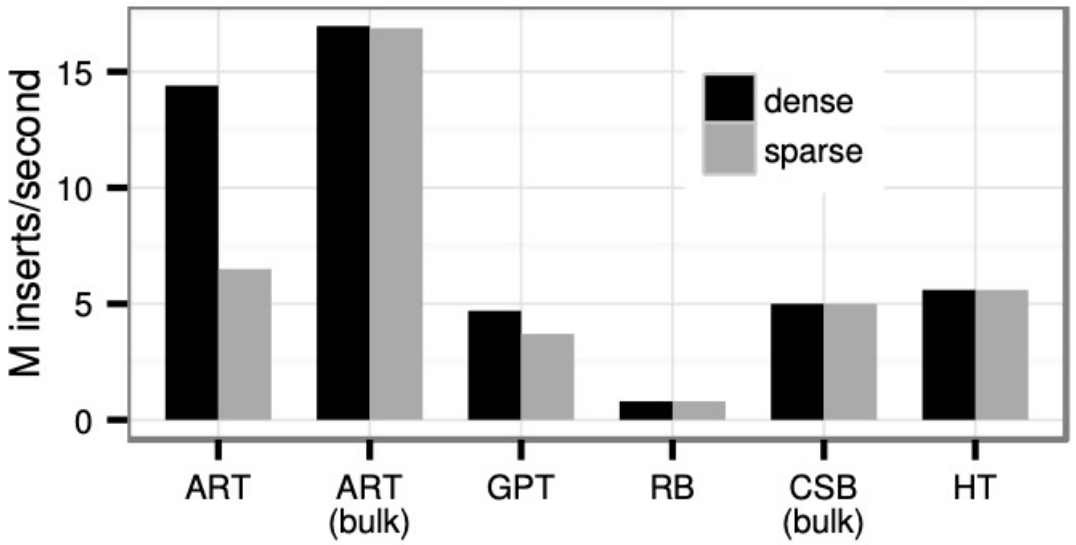


Fig. 14. Insertion of 16M keys into an empty index structure.

Evaluation—Single-Threaded Insert

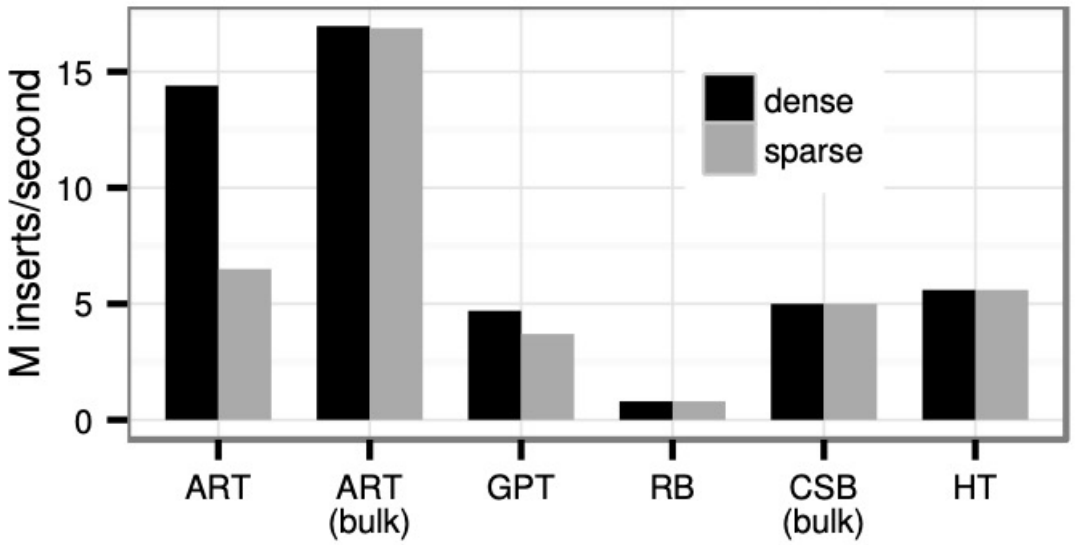


Fig. 14. Insertion of 16M keys into an empty index structure.

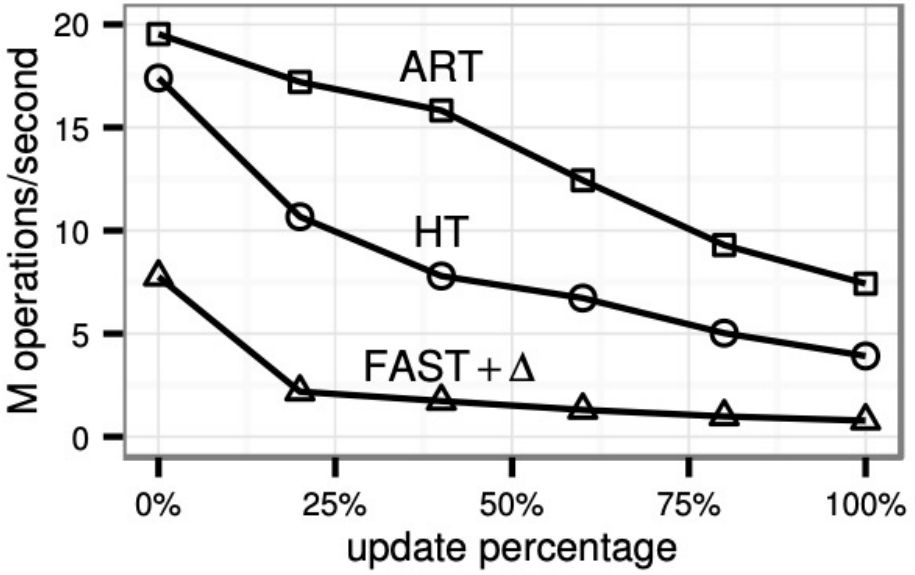


Fig. 15. Mix of lookups, insertions, and deletions (16M keys).

Evaluation – More Baselines

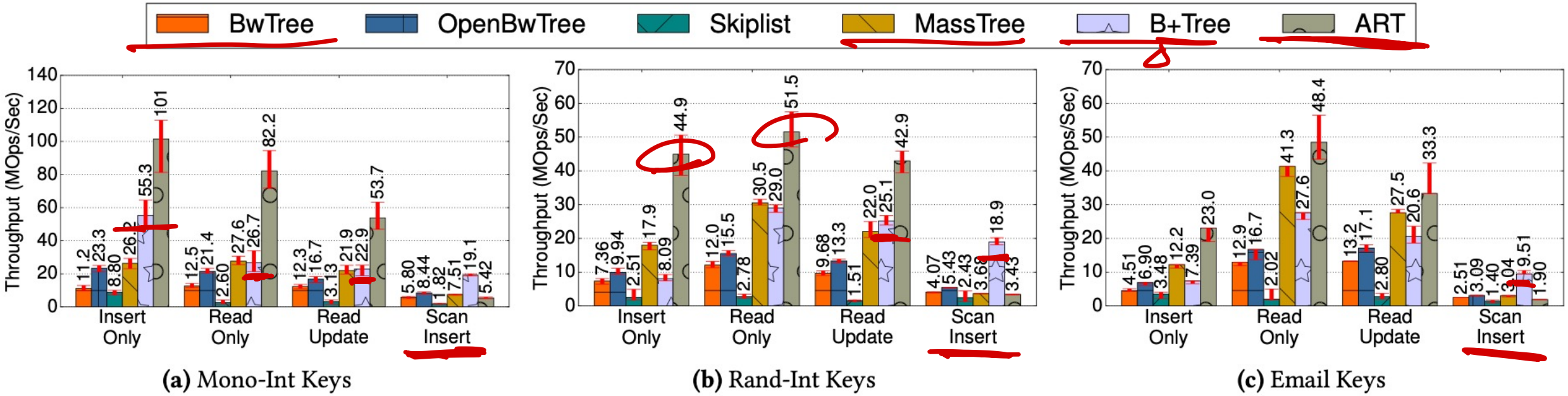
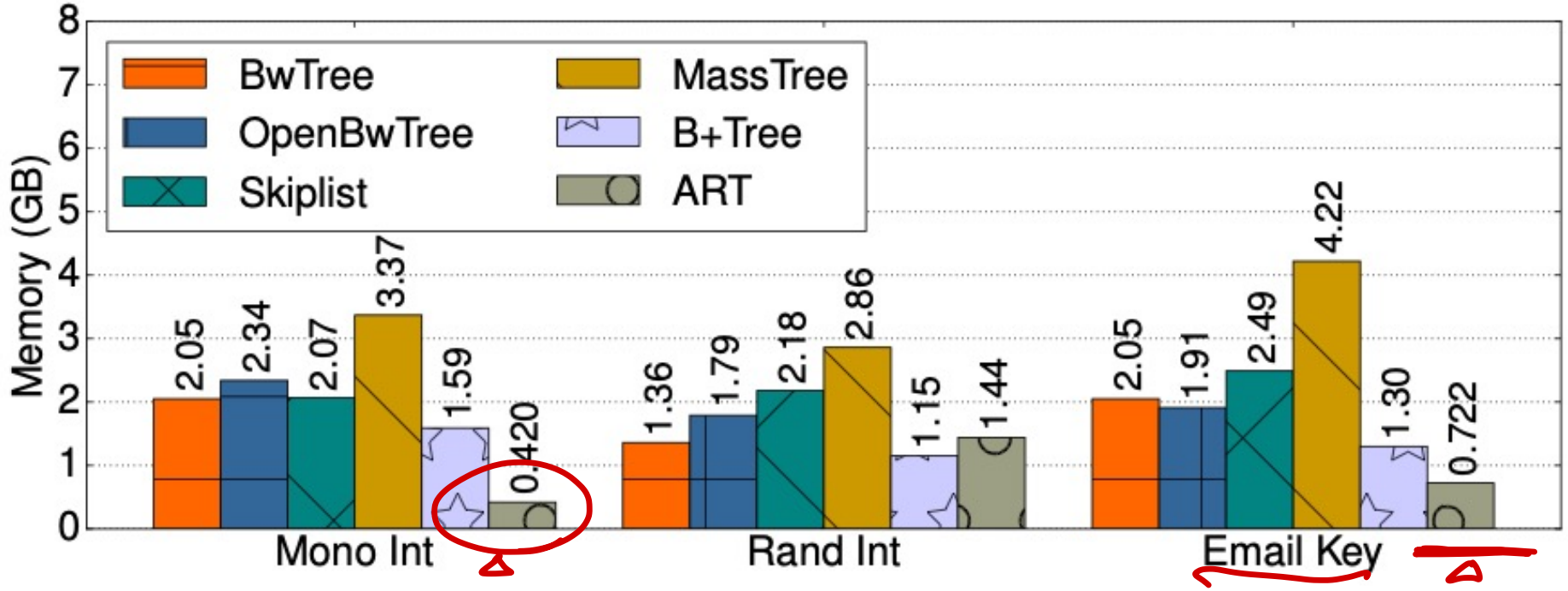


Figure 14: In-Memory Index Comparison (Multi-Threaded) – 20 worker threads. All worker threads are pinned to NUMA node 0.

* 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

ART performance on disk?

Tries used in real database systems?

Can ART leverage multicore hardware?

Why sort keys in Node4 and Node16?

Next Lecture

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