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

Xiangyao Yu
10/27/2020
The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases

Viktor Leis, Alfons Kemper, Thomas Neumann
Fakultät für Informatik
Technische Universität München
Boltzmannstrasse 3, 85748 Garching
<last name>@in.tum.de

Abstract—Main memory capacities have grown up to a point where most databases fit into RAM. For main-memory database systems, traditional binary search trees are no longer feasible because they are not optimally utilized on modern hardware. Traditional binary search trees are optimal for multi-core CPU caches. But as modern processors have evolved, the cost of CPU caches has increased, and the cost of main memory has decreased. This has led to a re-emergence of main memory indexing systems, which are now more efficient than traditional binary search trees.

1. INTRODUCTION

In recent years, the use of main memory has increased significantly, and traditional binary search trees have become less efficient. This is because modern CPUs have increased cache sizes, and the cost of main memory has decreased in comparison to the past. As a result, main memory indexing systems have become more efficient than traditional binary search trees.

More than 25 years ago, the T-trees [4] were proposed as an in-memory indexing structure. Unfortunately, the performance of these structures is not as good as traditional binary search trees, even on modern hardware. The reason is that the ever growing CPU cache sizes and the ever growing main memory have made the underlying assumption of uniform memory access time obsolete. This means that the efficiency of the structures is not as good as traditional binary search trees, even on modern hardware.

The reason is that the ever growing CPU cache sizes and the ever growing main memory have made the underlying assumption of uniform memory access time obsolete. This means that the efficiency of the structures is not as good as traditional binary search trees, even on modern hardware.

In conclusion, the use of main memory indexing systems is becoming more efficient than traditional binary search trees, even on modern hardware.
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**: 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
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

![Diagram](image)
Collapsing Inner Node

**Pessimistic**

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

![Diagram](image)

*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.
```plaintext
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

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)
```

19
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

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

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

* 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

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

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