Indexing for
Main-Memory data systems:
The Adaptive Radix Tree (ART)

Ivan Sinyagin
Memory Wall

Figure 1. Processor - Memory Performance Imbalance [2]
Why indexes?
Best data structure

O(1)?
Binary Search !
Binary Search

- Cache utilization is low
- Only first 3-5 cache lines have good temporal locality
- Only the last cache line has spacial locality
- Updates in a sorted array are expensive
Trees
T-tree

- Sorted array split into balanced BST with fat nodes (~ cache lines)
- Better than RB/AVL
- Updates faster, but still expensive
- Similar to BS: useless data movement to CPU (useful only min and max)
- Developed in mid 80s and still(!) used in many DBMS
B+ tree
B+ tree

- Fanout => minimize random access by shallowing the tree
- Keys fit into a cache line
- Increased cache utilization (all keys are useful)
- 1 useful pointer
- Pipeline stalls - conditional logic
- Still expensive updates: splitting & rebalancing
CSB+ tree
C SB+ tree

• ~ 1999-2000

• Improved space complexity

• Great cache line utilization: keys + 1 pointer

• Node size ~ cache line

• Update overhead - more logic to balance
Can we do better?

- Less conditional logic
- Cheap updates: no rebalancing, no splitting
- Preserve order => tree
- Preserve few random accesses (low height)
- Preserve cache line utilization
- Preserve space complexity
Tries
Radix Tree

Example: Short k=16, k'=4 \rightarrow h=4
INSERT key=107, payload="value3"

key = 107 0000 0000 0110 1011
path 0 0 6 11

Implicit keys
Space complexity
Radix Tree span

• k bits keys => k/s inner levels and $2^s$ pointers
• 32 bit keys & span=1 => 32 levels & 2 pointers
• 32 bit keys & span=2 => 16 levels & 4 pointers
• 32 bit keys & span=3 => 11 levels & 8 pointers
• 32 bit keys & span=4 => 8 levels & 16 pointers
• 32 bit keys & span=8 => 4 levels & 256 pointers
Adaptive Radix Tree

Idea - node resizing based on capacity

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

- 1M keys
- ART height ~ B+ tree
Adaptive nodes

N256 implicit keys

```
typedef struct {
    art_node n;
    art_node *children[256];
} art_node256;
```

N4 & N16 explicit keys

```
typedef struct {
    art_node n;
    unsigned char keys[16];
    art_node *children[16];
} art_node16;
```

N48 indirection index

```
typedef struct {
    art_node n;
    unsigned char keys[256];
    art_node *children[48];
} art_node48;
```
Algorithms

- Search: conditional logic only within a cache line
- Insert: no rebalancing/splitting, possible resize
- Delete: no rebalancing/splitting, possible shrink
- Bulk load: builds ART while performing radix sort
- Code: paper + https://github.com/armon/libart
ART Optimizations

Path Compression & Leaf Expansion

Fig. 6. Illustration of lazy expansion and path compression.
Path compression
Binary-Compatible keys

- Strings have lexicographic order
- Natural numbers have bit order
- Integers: negative 2-complement ints
- Required transformations before storing in ART: floats, unicode, signed, null, composite
Evaluation

• Micro benchmark (removed path compression) against
  • CSB+ tree (~2001)
  • FAST (static array-based tree index) (2010)
  • GPT (~2009)
  • RB tree (textbook)
  • Hash Table (chained, textbook)
• HyPer: OLPT TPC-C
Dense vs Sparse keys

- Sparse (each bit may equally be 0 or 1)
- Dense (0, 1, 2 … n) - high N256 space utilization
Random search performance

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

- L3 Misses: 0 in 65K
- Misp. Branches: 0 in ART dense keys (N265)
Multithreaded search and software pipelining

- FAST speed-up 2.5x (computationally intensive)
- ART speed-up 1.6x (4-level tree)
- HT speed-up 1.2x (2-level tree)
Skewed search

- ART: adjacent items in the same subtree
- HT: adjacent items in different buckets

**Fig. 12.** Impact of skew on search performance (16M keys).
Round-robin dense search: cache size

- ART: no eviction; fewer misses
- HT randomly distributes; more misses

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

- Radix Tree: cheap inserts in general
- Adaptive nodes overhead ~20%
- Dense keys are cache-friendly: fully occupied N256 => less conditional logic
- Bulk loading: transforms sparse into dense
Random workload: lookup & update

- Update in ART: same subtree
- Update in HT: different buckets

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

- HyPer: indexes ~ performance (no buffer management, no locking, no latching)
- TPC-C: skewed data, 46% updates
Impact of optimizations
More HT problems

- all keys are randomly distributed
- dense search cannot use temporal locality
- updates cannot use temporal locality
Concerns

- HT from a textbook. Partitioning?
- CSB+ Tree implementation ~ 2001
- CSB+ Tree crashed with 256M keys - why?
- Space utilization for sparse keys
- Few tests that used sparse keys
Proof & experiment
Worst case key space consumption

• Proof by induction for the current setup: <= 52

<table>
<thead>
<tr>
<th>#</th>
<th>Relation</th>
<th>Cardinality</th>
<th>Attribute Types</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>item</td>
<td>100,000</td>
<td>int</td>
<td>8.1</td>
</tr>
<tr>
<td>2</td>
<td>customer</td>
<td>150,000</td>
<td>int,int,int</td>
<td>8.3</td>
</tr>
<tr>
<td>3</td>
<td>customer</td>
<td>150,000</td>
<td>int,int,vchar(16),vchar(16),TID</td>
<td>32.6</td>
</tr>
<tr>
<td>4</td>
<td>stock</td>
<td>500,000</td>
<td>int,int</td>
<td>8.1</td>
</tr>
<tr>
<td>5</td>
<td>order</td>
<td>22,177,650</td>
<td>int,int,int</td>
<td>8.1</td>
</tr>
<tr>
<td>6</td>
<td>order</td>
<td>22,177,650</td>
<td>int,int,int,int,TID</td>
<td>24.9</td>
</tr>
<tr>
<td>7</td>
<td>orderline</td>
<td>221,712,415</td>
<td>int,int,int,int</td>
<td>16.8</td>
</tr>
</tbody>
</table>
References

(B) A study of index structures for main memory database management systems
T. J. Lehman and M. J. Carey
International Conference on Very Large Databases (VLDB), 1986

(B) Cache conscious indexing for decision-support in main memory
J. Rao and K. Ross
International Conference on Very Large Databases (VLDB), 1999

(B) Making B+-Trees Cache Conscious in Main Memory
J. Rao and K. Ross, 2000

(B) Node Compression Techniques based on Cache-Sensitive B+-tree
Rize Jin, Tae-Sun Chung, 2010

(P) The Adaptive Radix Tree: ARTful indexing for main-memory databases.
Viktor Leis, Alfons Kemper, Thomas Neumann
International Conference on Data Engineering (ICDE), 2013