Efficient Indexing in Main Memory: Adaptive Radix Tree (ART)

Overcoming Index Structure Performance Bottlenecks

Changxuan Fan Boyang Liu Chunhao Bi

Current Problem

- Main memory capacities have grown, enabling most databases to fit into RAM.
- Binary search trees: inefficient due to hardware advancements.
- Hash tables: fast but only support point queries.

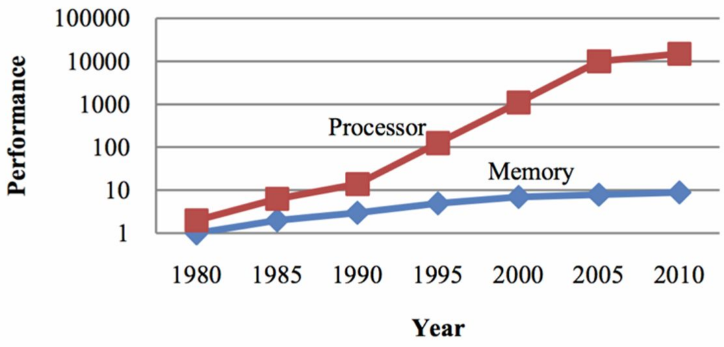
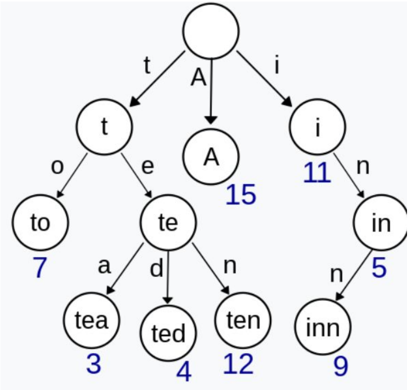
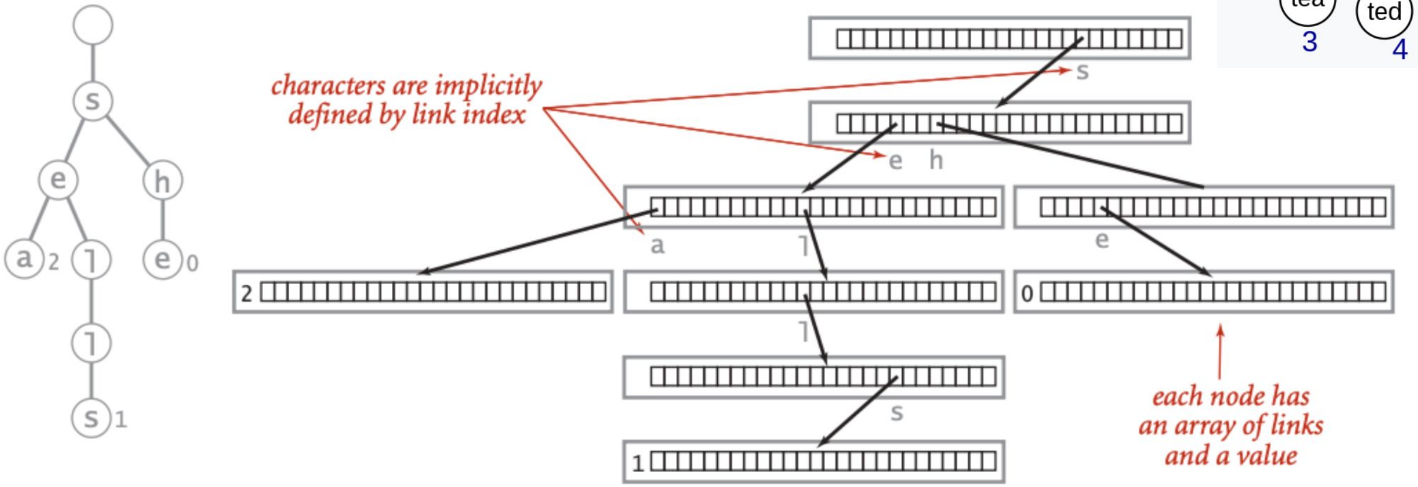


Figure 1. Processor - Memory Performance Imbalance [2]

Prefix Tree (Trie)

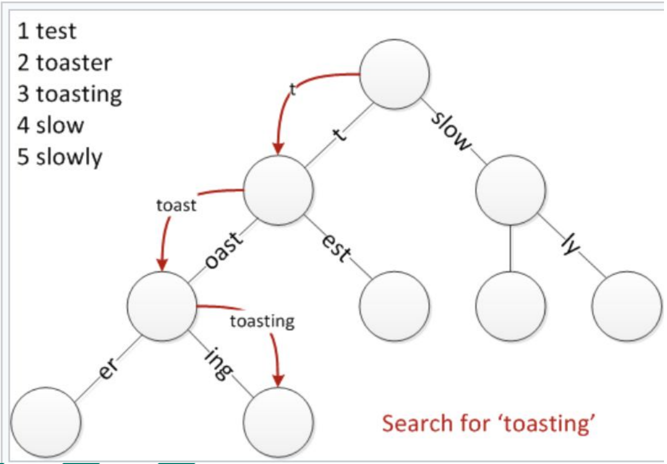
- Unlike a binary search tree, nodes in the trie do not store their associated key.
- Instead, a node's position in the trie defines the key with which it is associated.

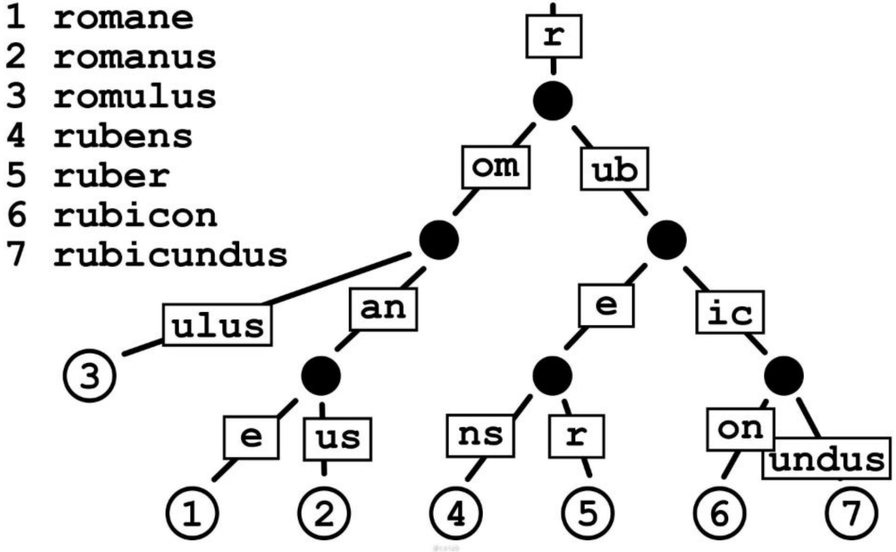




Radix Tree (Background)

- Unlike Prefix Tree, each node that is the only child is merged with its parent.





ART (adaptive radix tree)

- ART, offers efficient indexing, surpassing traditional structures and supporting insertions/deletions.
- ART maintains sorted order, enabling additional operations like range scan and prefix lookup.

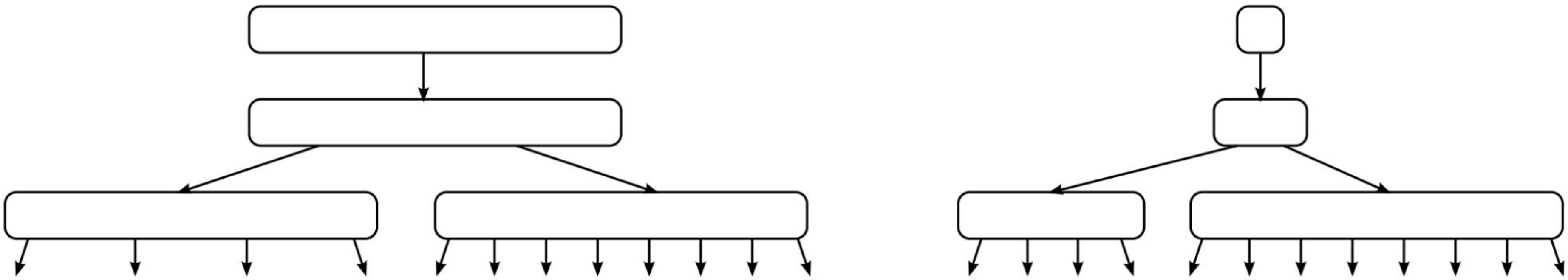
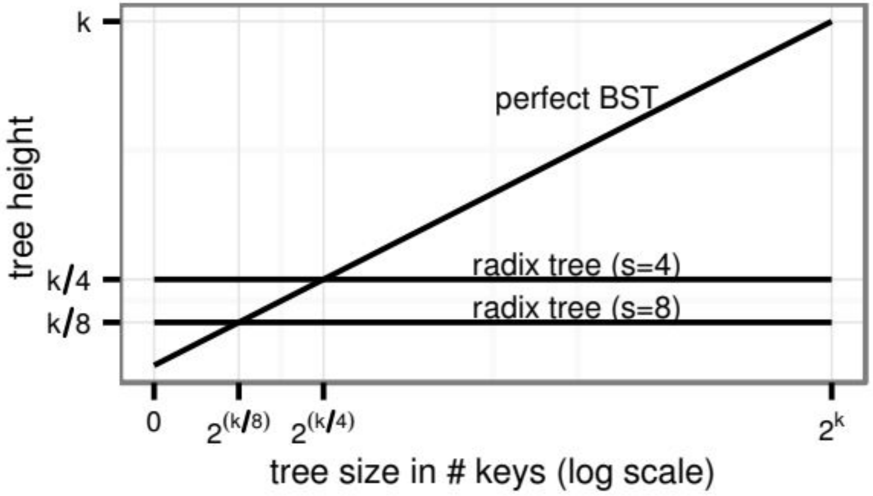


Fig. 4. Illustration of a radix tree using array nodes (left) and our adaptive radix tree ART (right).

Radix trees vs. comparison-based trees:

- Height depends on key length, not on the number of elements.
- No rebalancing operations required.
- Keys stored in lexicographic order.
- Inner nodes map partial keys, leaf nodes store values.
- Complexity comparison: O(k log n) vs. O(k).



Adaptive Nodes

- Desirable large span vs. excessive space consumption:
  - Trade-off illustrated with different span values.
  - Adaptive Radix Tree (ART) introduces adaptivity in node sizes.
- Adaptive node illustration:
  - Maintains tree structure while adjusting node sizes.
- Efficient support for incremental updates:
  - Small number of node types with different fanouts.
  - Replacement of nodes based on capacity and underfull conditions.

Inner Node

- Inner nodes map partial keys to child pointers.
- Four data structures with different capacities used internally.

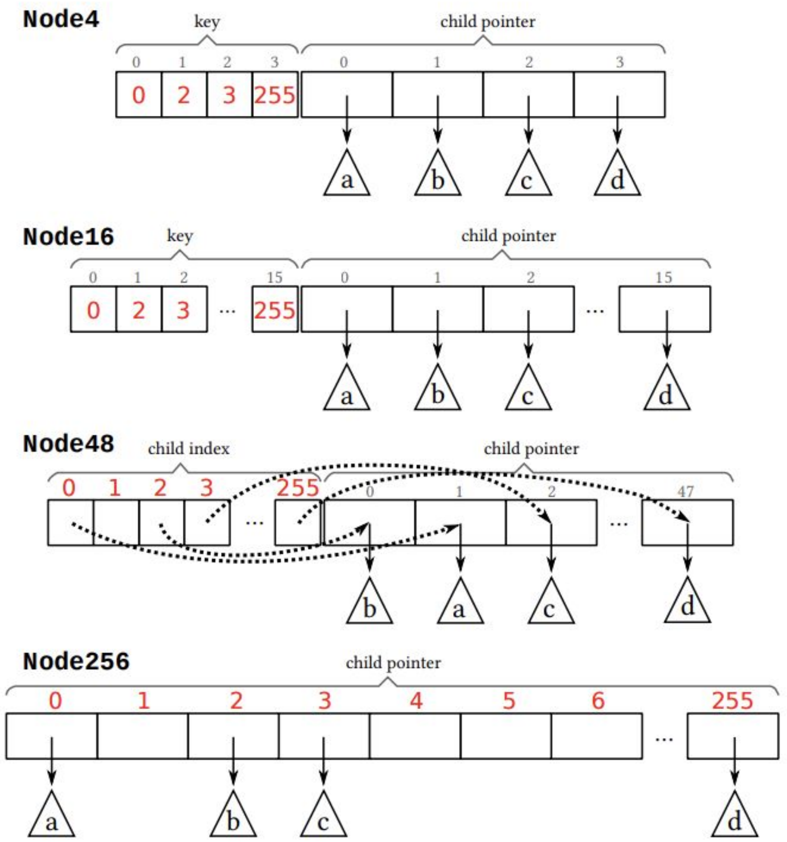


Fig. 5. Data structures for inner nodes. In each case the partial keys 0, 2, 3, and 255 are mapped to the subtrees a, b, c, and d, respectively.

Leaf Node

Structure of Leaf Nodes:

- Discussion on storing values associated with keys.
- Different methods for storing values:
  - Single-value leaves
  - Multi-value leaves
  - Combined pointer/value slots

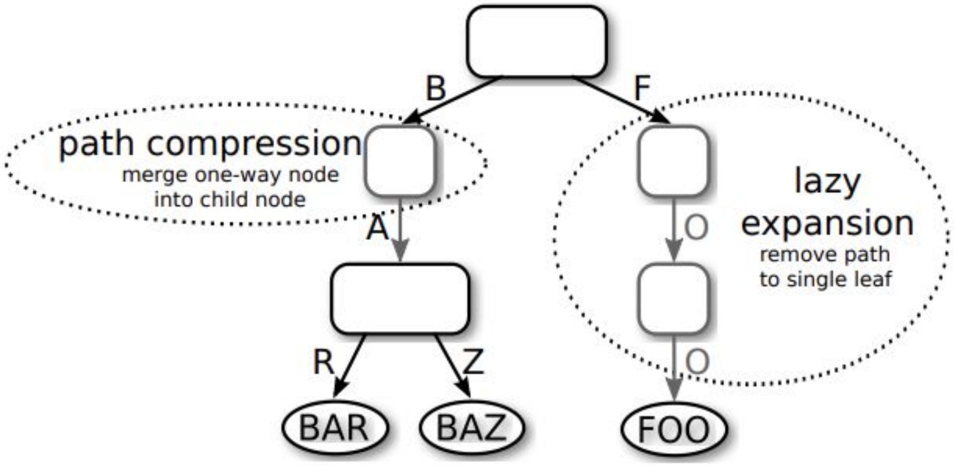


Fig. 6. Illustration of lazy expansion and path compression.

Now it's time to create an **ART** from a naive radix tree!

Let's first **compress** the tree!

Optimization - Node Collapse

1. Lazy Expansion
   - Inner nodes are only created if they are required to distinguish at least two leaf nodes
2. Path Compression
   - Removes all inner nodes that have only a single child

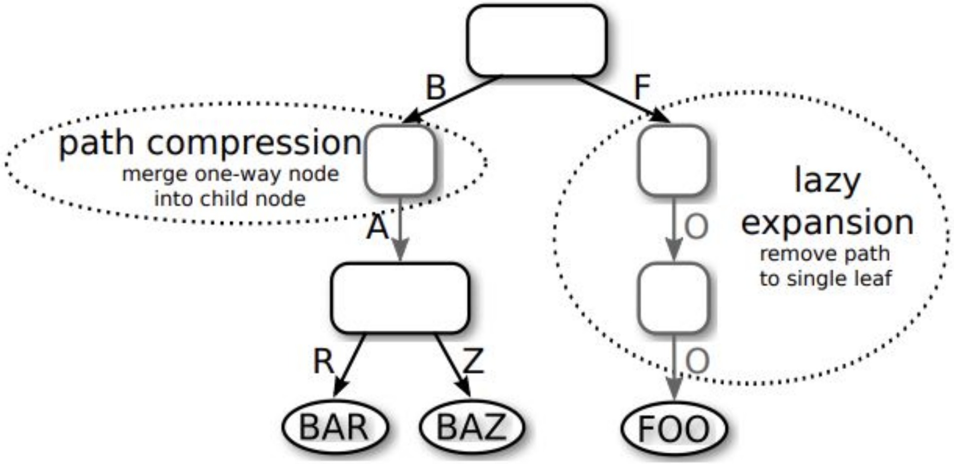
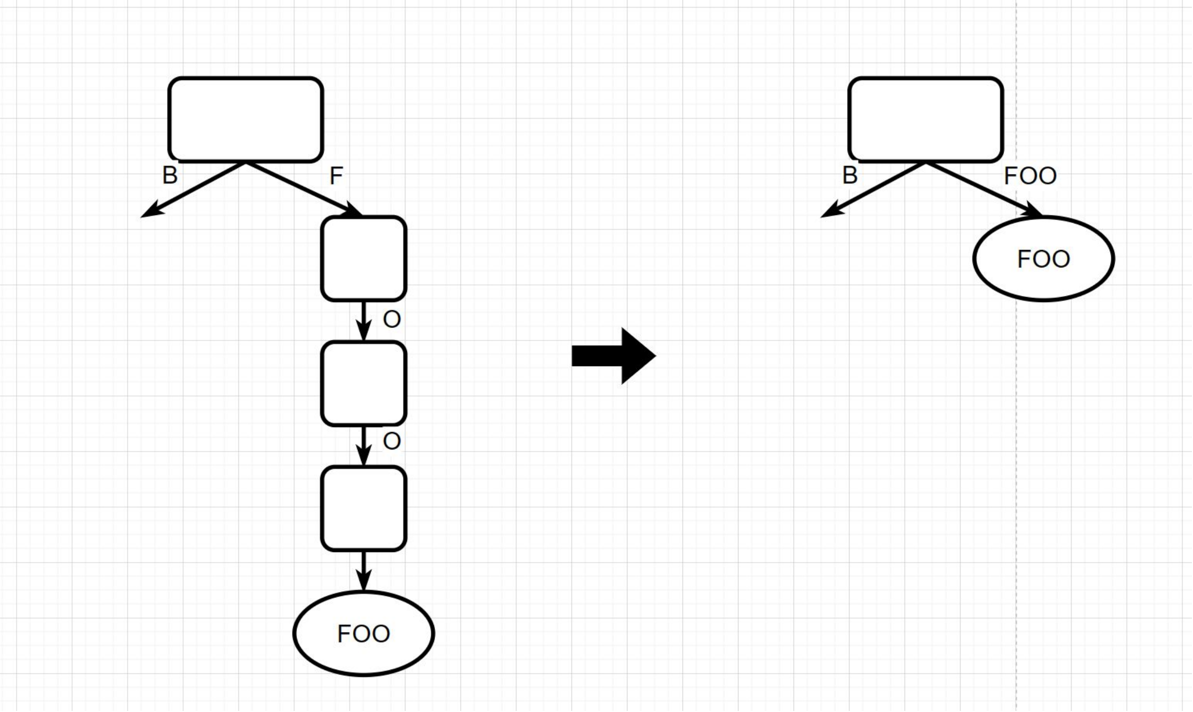
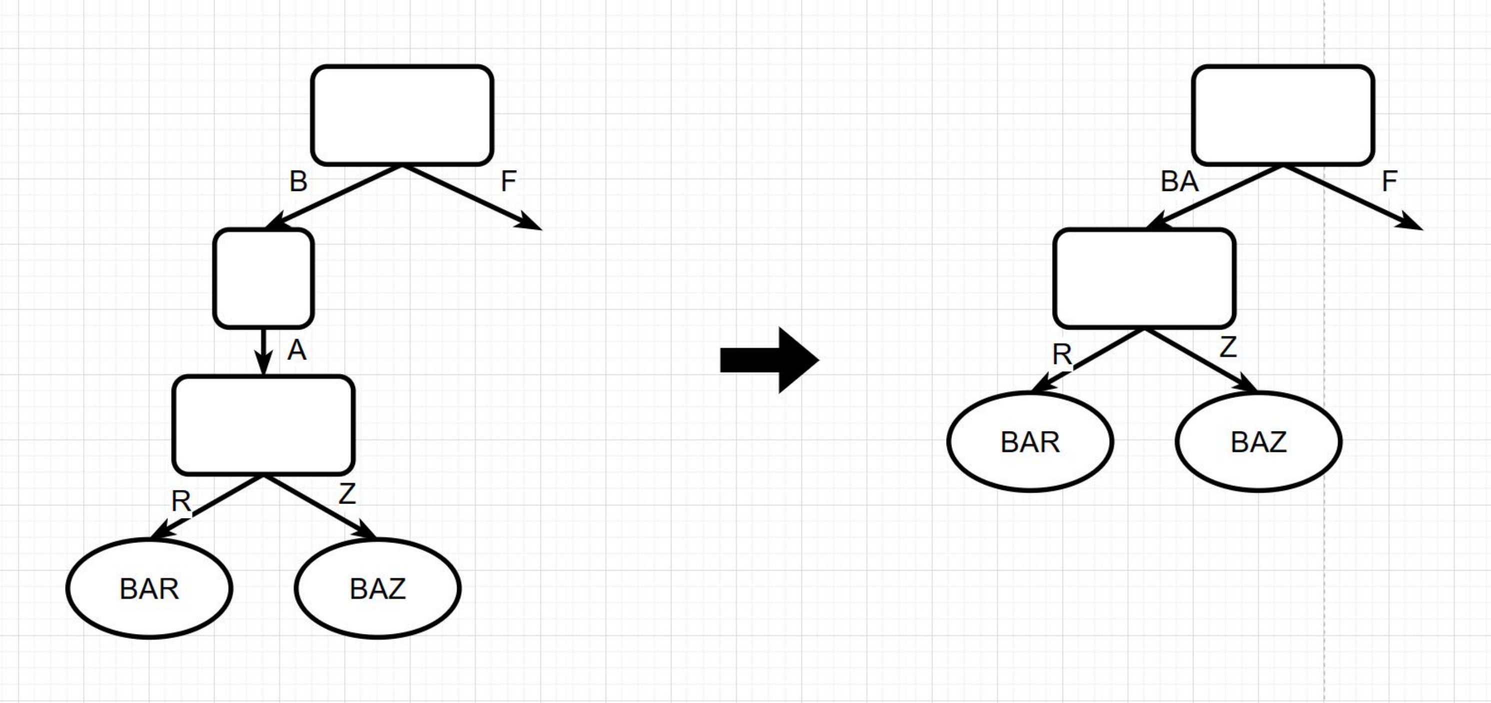


Fig. 6. Illustration of lazy expansion and path compression.

Lazy Expansion



Path Compression



We’ve done the **compression**, what’s next?

Let's **search** first!

Search

```
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.

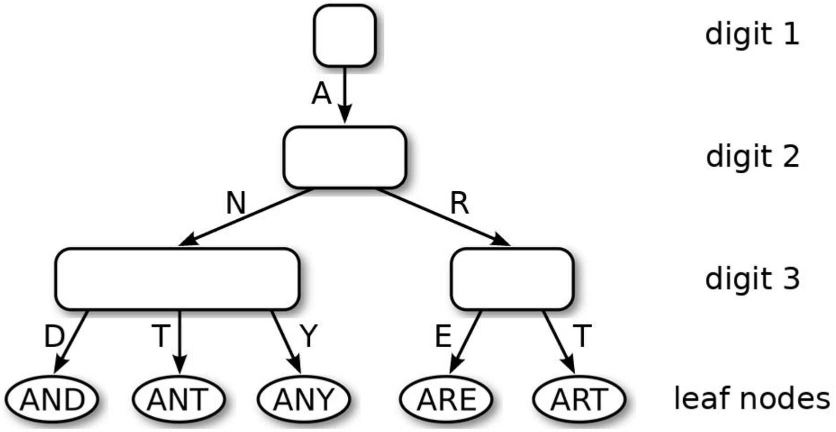
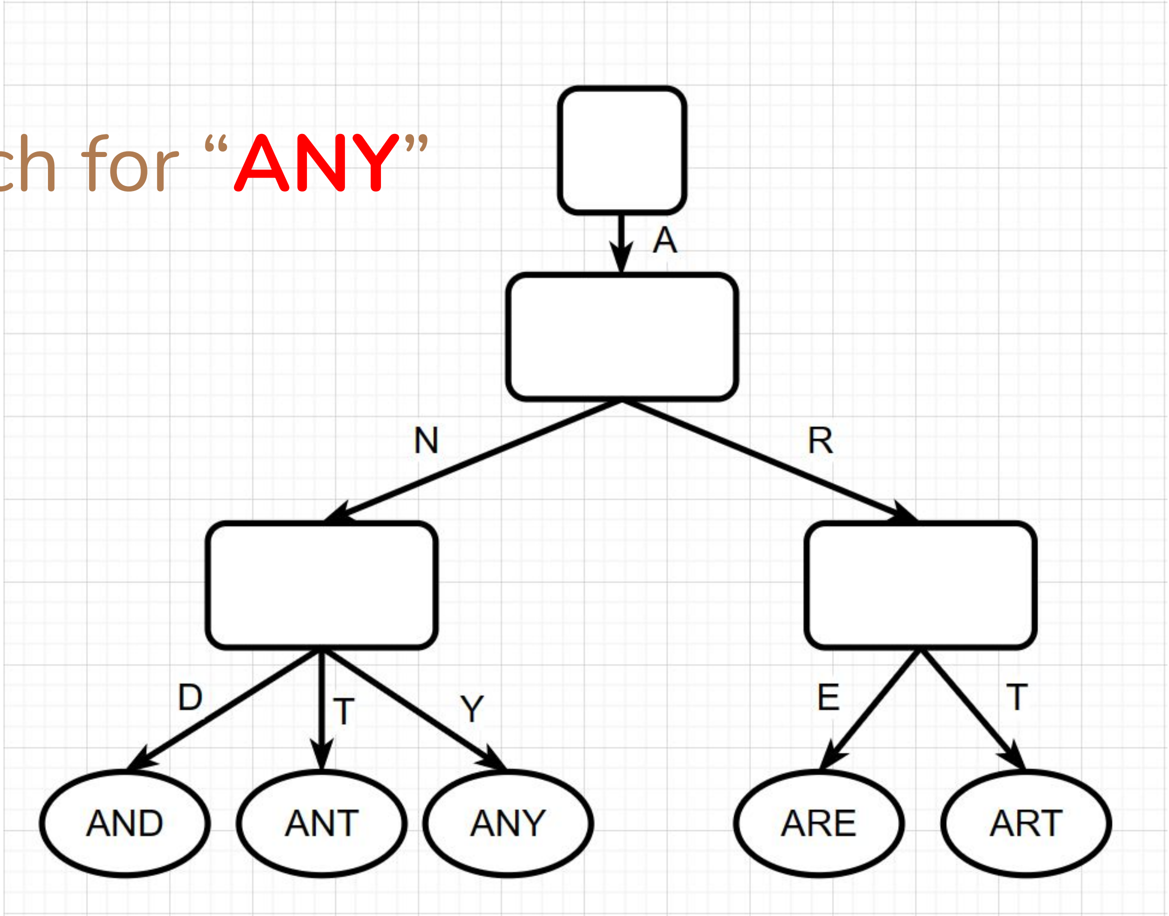
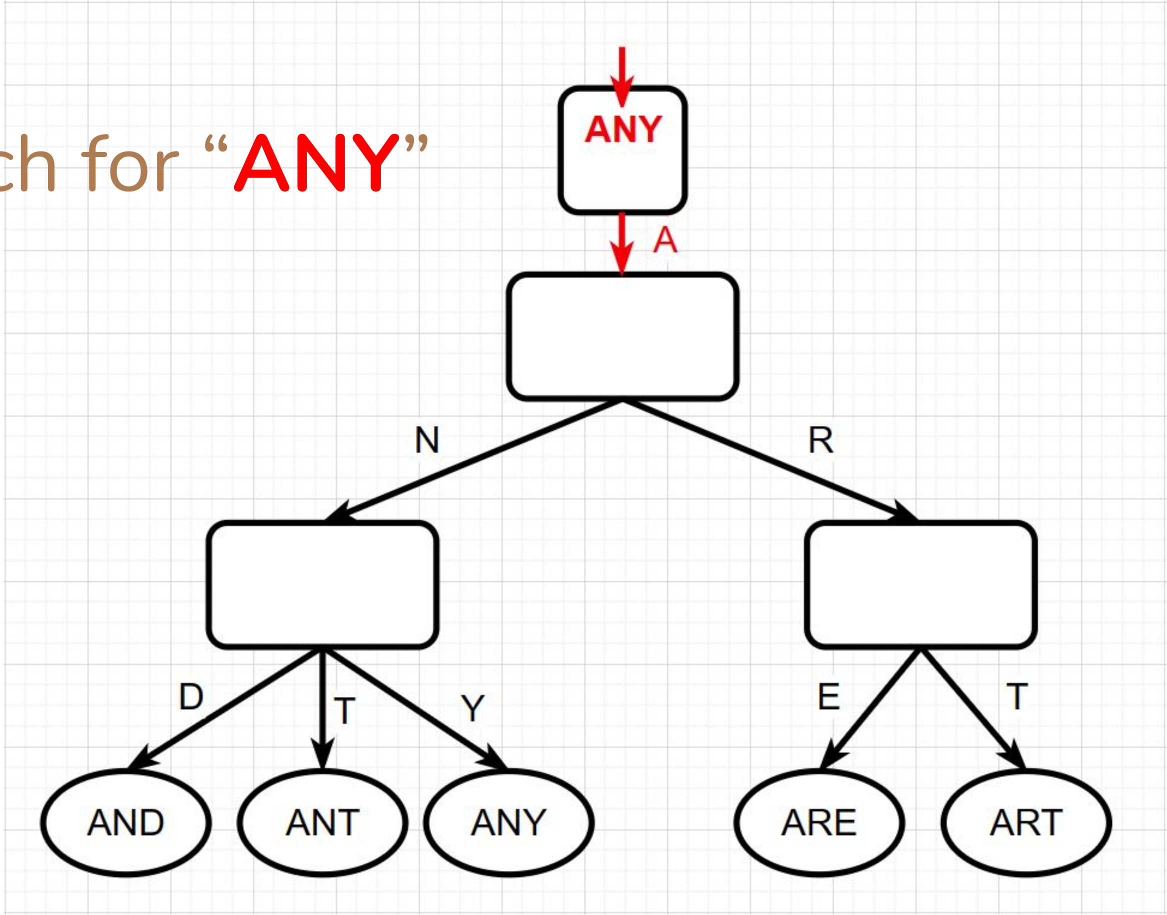


Fig. 1. Adaptively sized nodes in our radix tree.

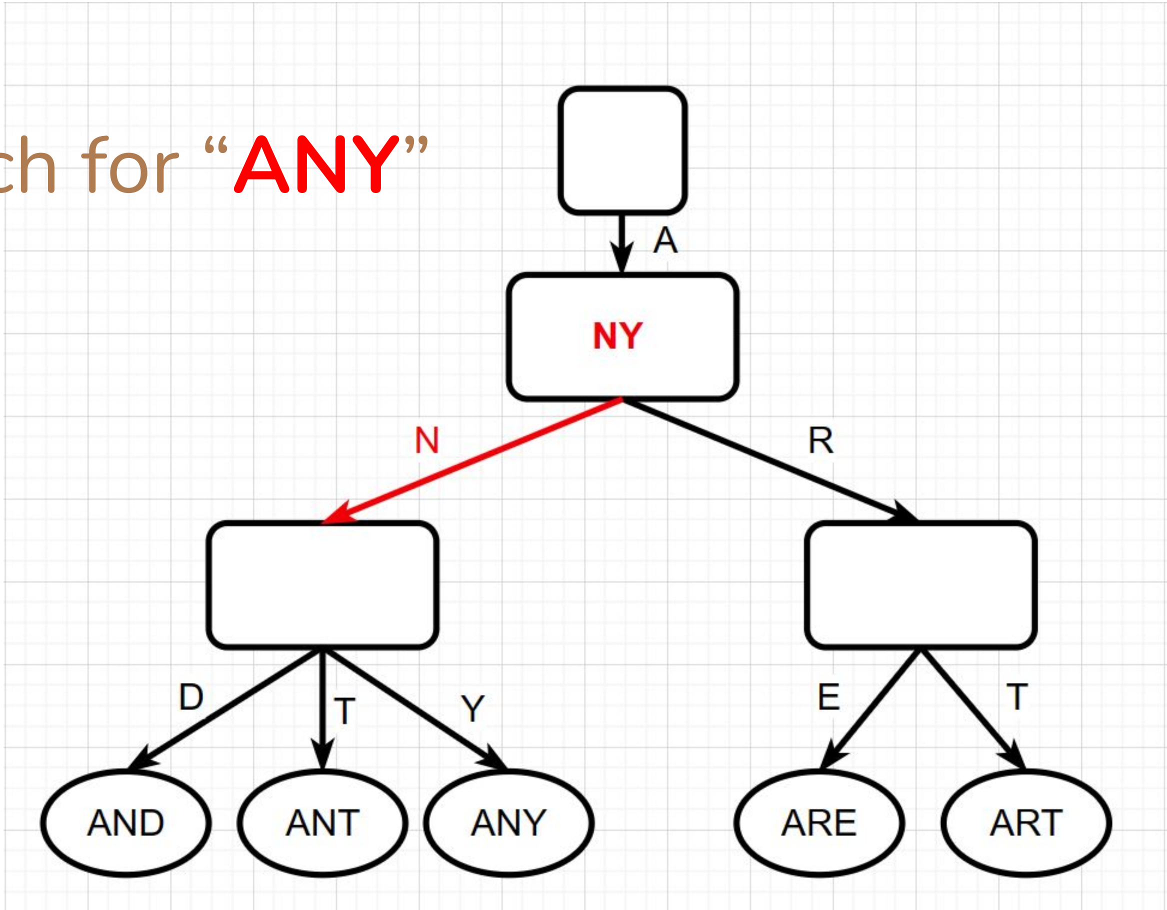
Search for “ANY”



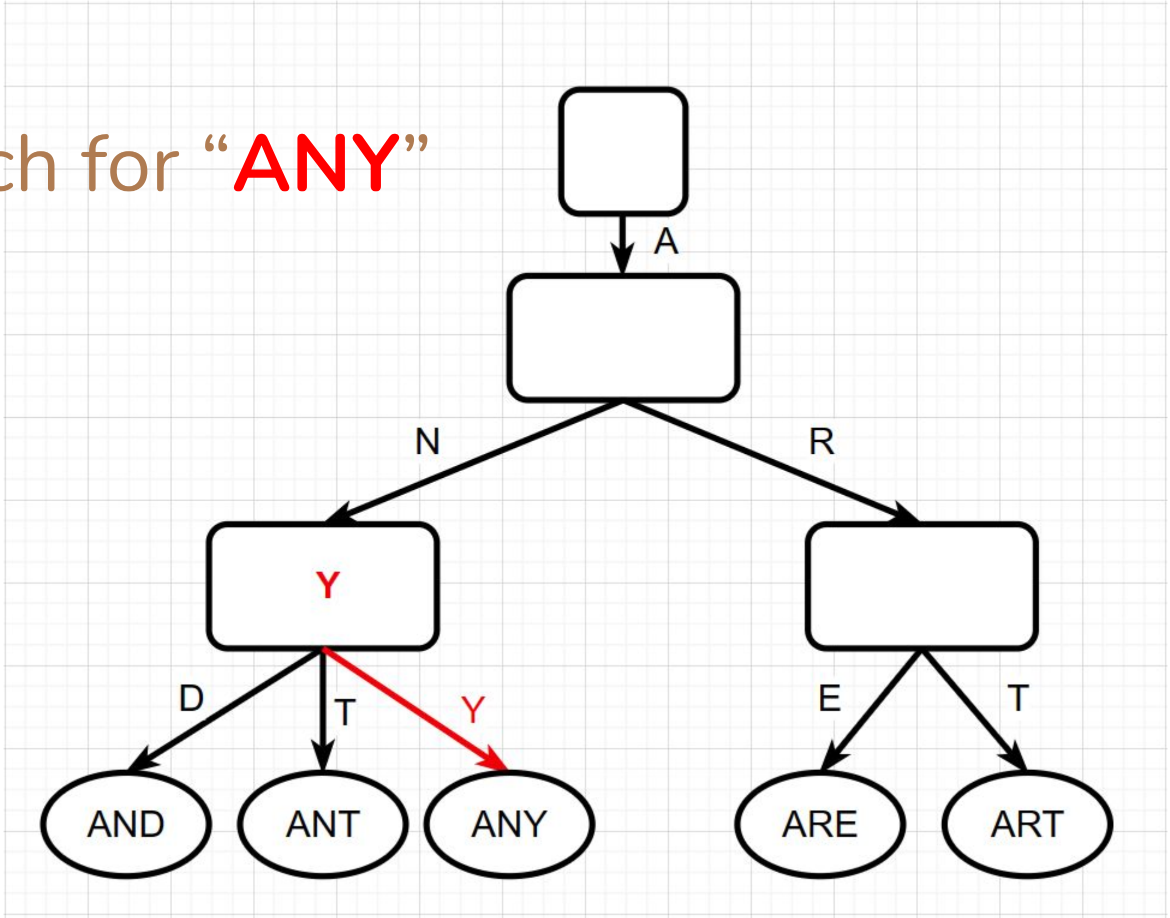
Search for "**ANY**"



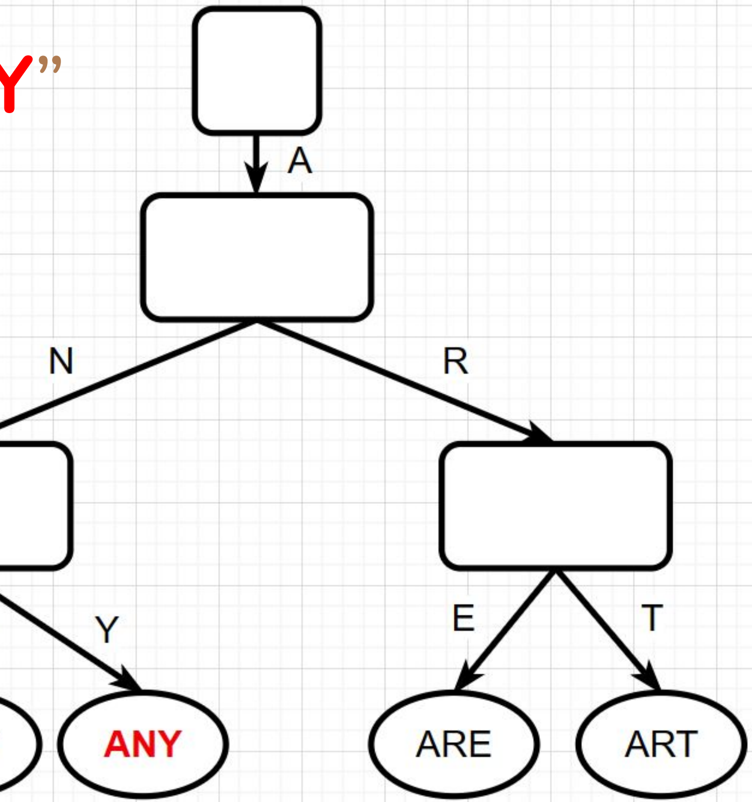
Search for “ANY”



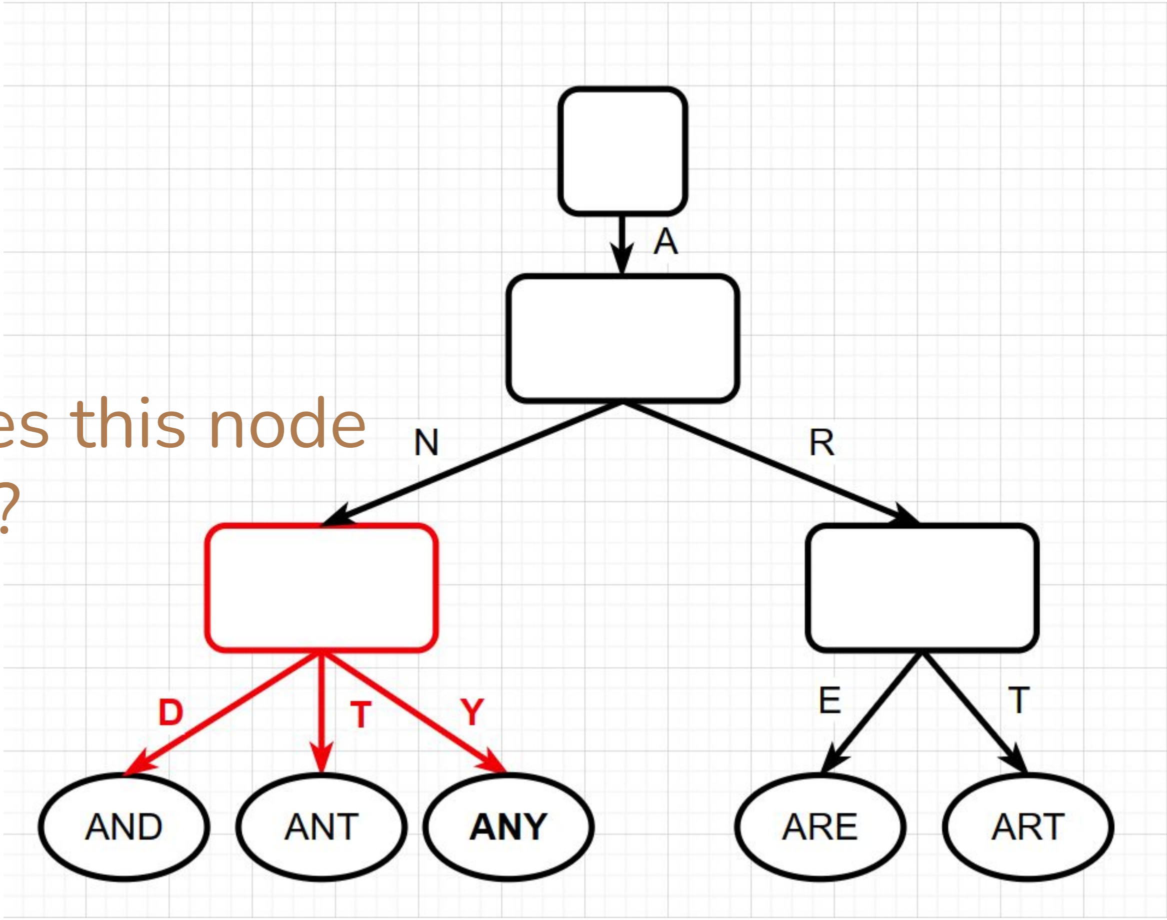
Search for “ANY”

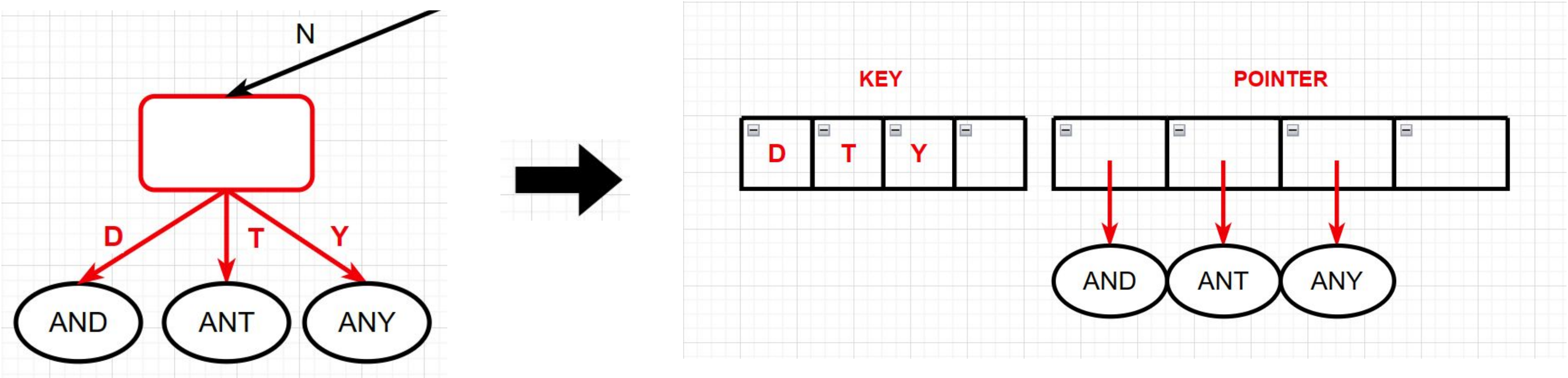


Search for "**ANY**"



How does this node look like?





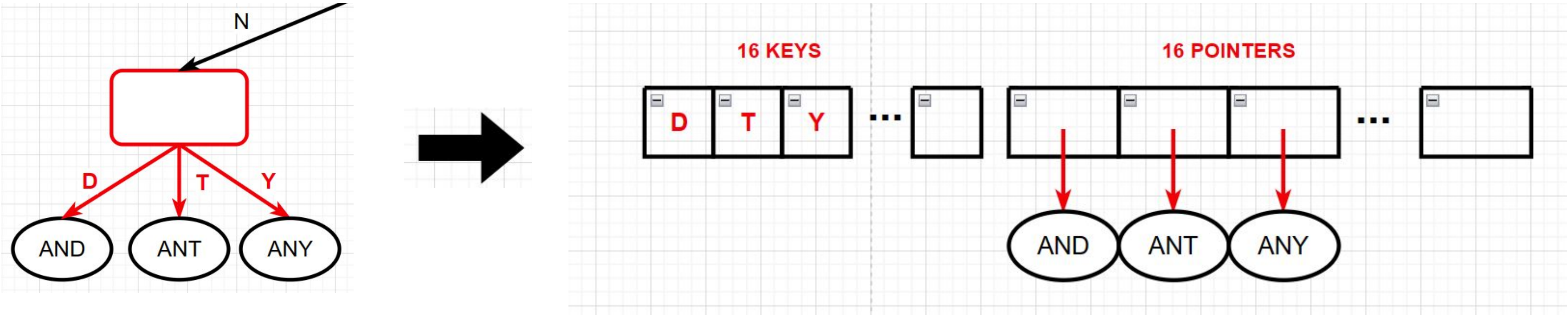
This is a **Node4**

Compare each key one by one

What if we have more than **4** values?

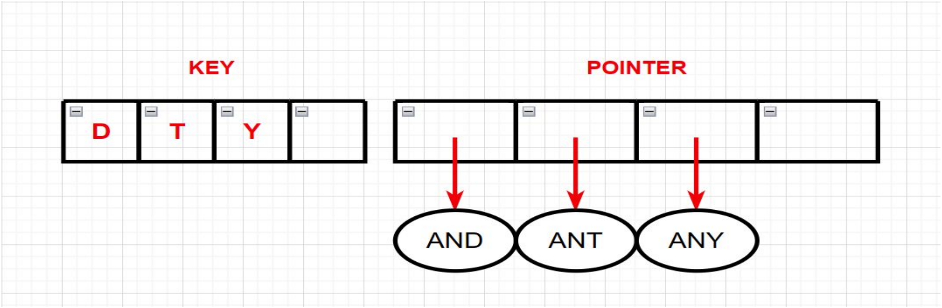
Node4 store only **4** values!

We use **Node16**!

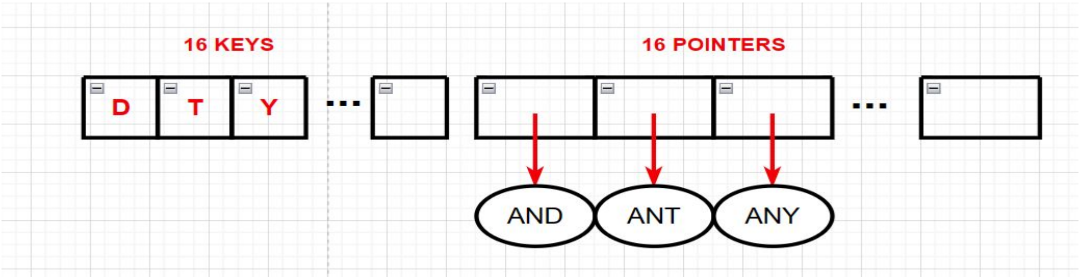


This is a **Node16**

Still compare each key one by one?



Node4

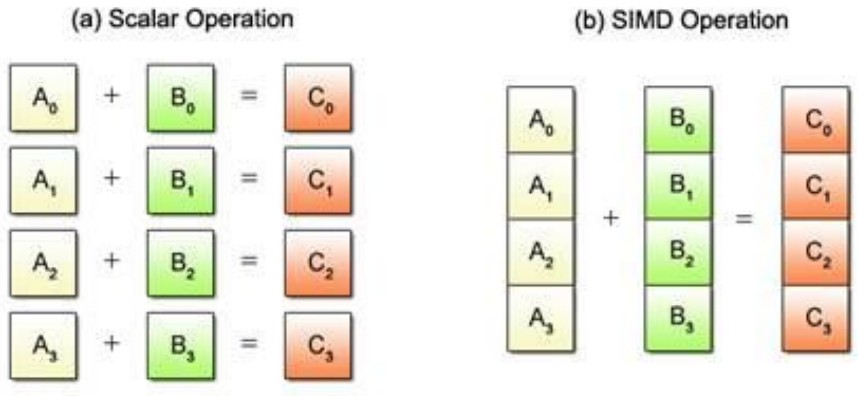


Node16

What's the difference between **Node4** and **Node16**? Only size?

SIMD - Single instruction, multiple data

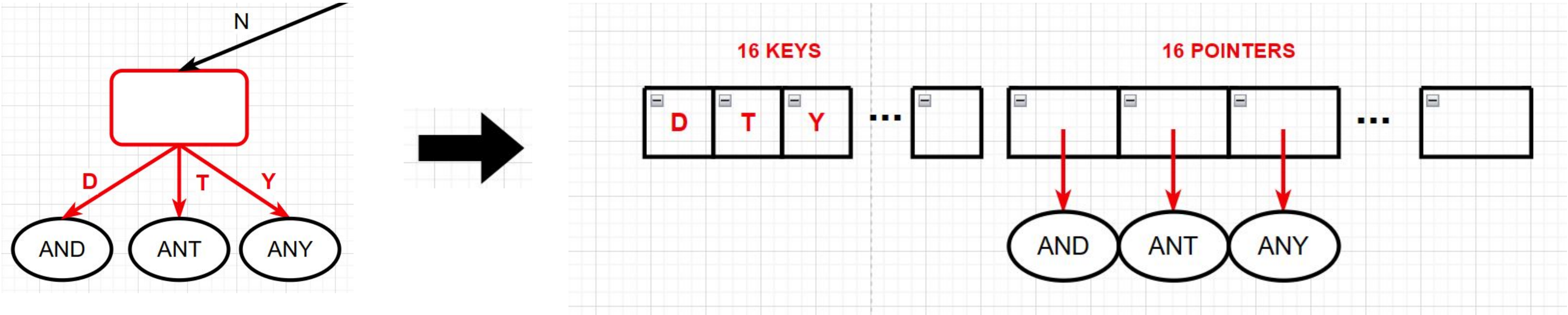
- Supported by most modern computers
- Compute multiple data with one run




```
__m128i _mm_cmpeq_epi8(__m128i a, __m128i b)
```

Compares the 16 signed or unsigned 8-bit integers in `a` and the 16 signed or unsigned 8-bit integers in `b` for equality.

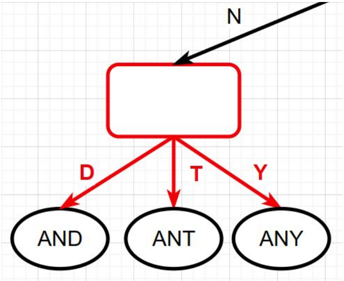
R0	R1	...	R15
`(a0 == b0) ? 0xff : 0x0`	`(a1 == b1) ? 0xff : 0x0`	...	`(a15 == b15) ? 0xff : 0x0`

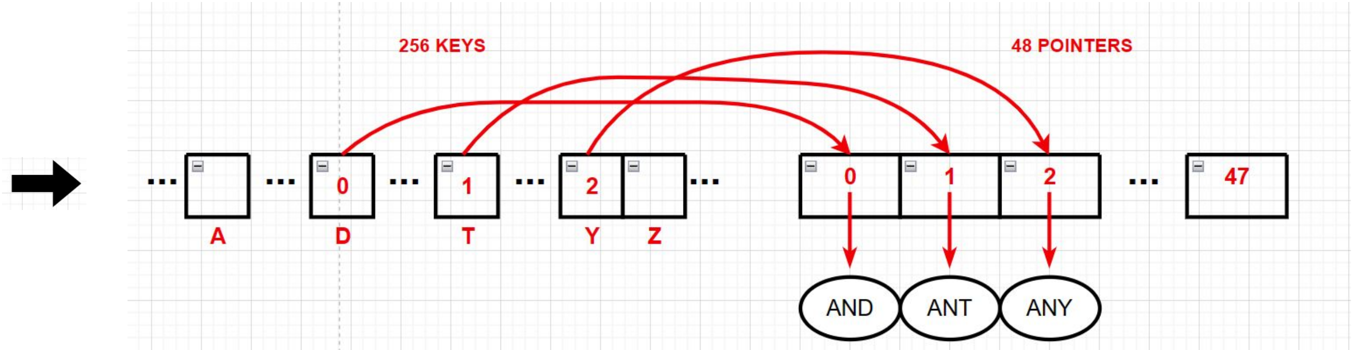


We can compare keys in parallel!

(Or use binary search if not supported)

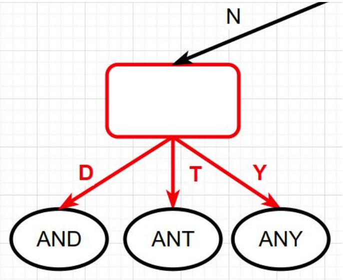
More keys? Can’t compare 48 keys in parallel!

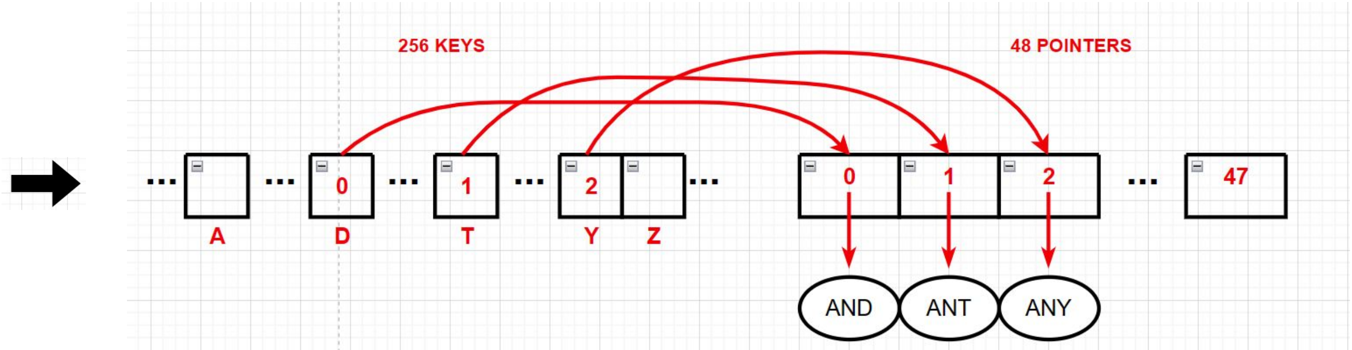




This is a **Node48**

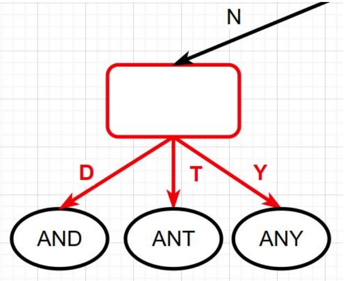
At most 48 out of 256 keys have index, rest are EMPTY!!

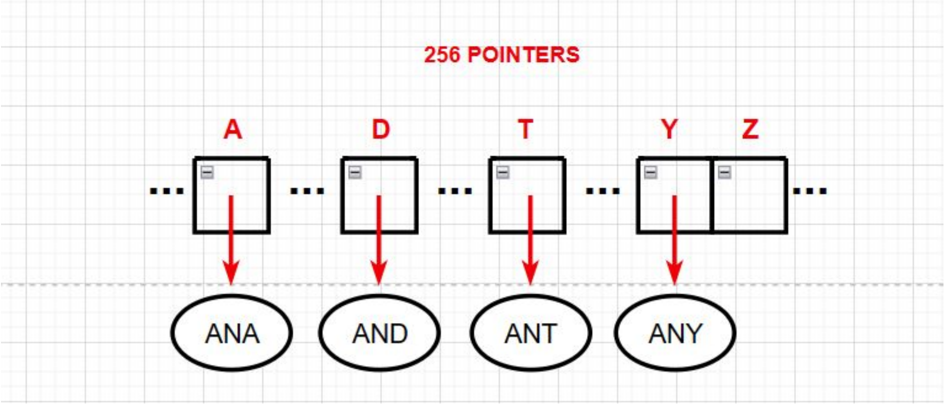




Why 256 keys?

Because each key 1 byte = 8 bits = 256 values





This is a **Node256**

Every key can have a pointer

Search

- Node lookup

1. **Node4** - traverse
2. **Node16**
    - parallel(or binary search)
3. **Node 48**
    - return node[index[byte]]
4. **Node256**
    - return node[key]
5. **Time Complexity? Almost O(1)!**

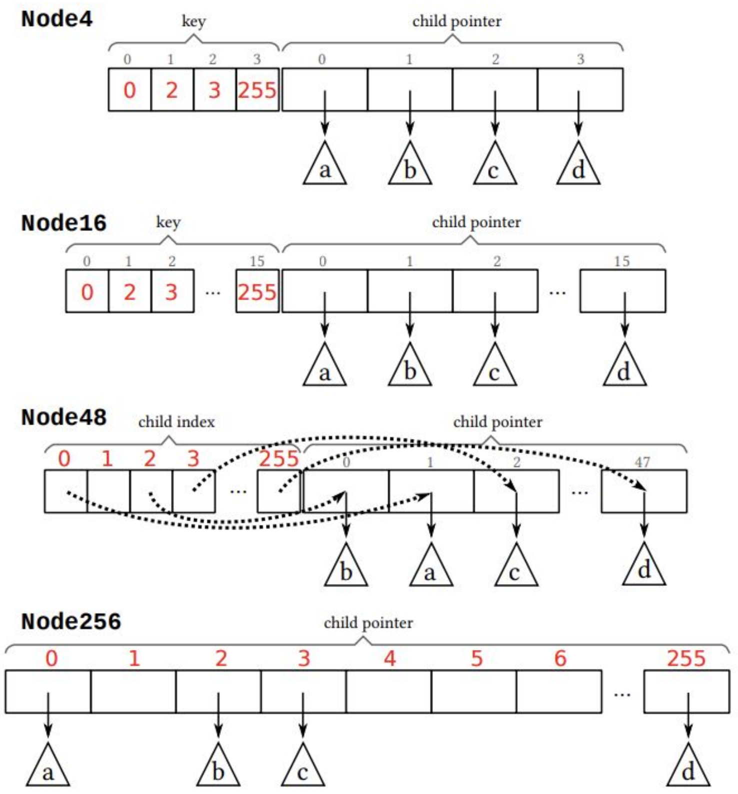
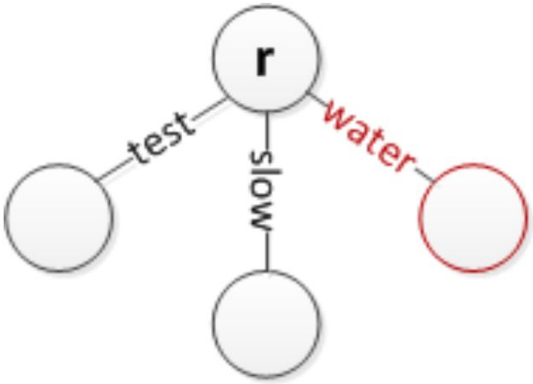


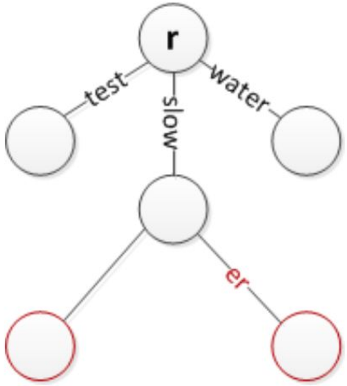
Fig. 5. Data structures for inner nodes. In each case the partial keys 0, 2, 3, and 255 are mapped to the subtrees a, b, c, and d, respectively.

Insert

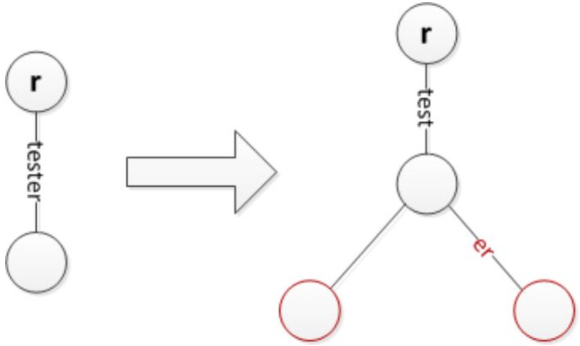
1. Add Leaf
2. Postfix
3. Prefix
4. Split
5. Node grow



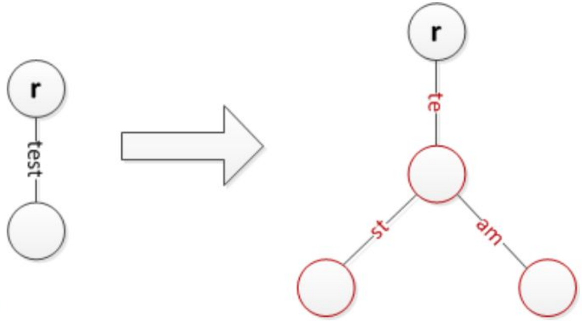
Insert 'water' at the root



Insert 'slower' while keeping 'slow'



Insert 'test' which is a prefix of 'tester'



Insert 'team' while splitting 'test' and creating a new edge label 'st'

Delete

- Symmetrical to insertion.
- The leaf is removed from an inner node, which is shrunk if necessary. (Path Compression)
- If a node now has only one child, it is replaced by its child(Lazy Expansion).

Bulk loading

- Using the first byte of each key the key/value pairs are radix partitioned into 256 partitions and an inner node of the appropriate type is created.
- Before returning that inner node, its children are created by recursively applying the bulk loading procedure for each partition using the next byte of each key.

How does ART reduce space consumption?

Reduce space consumption

- **Optimization** (Lazy Expansion + Path Compression)
- **Adaptive Nodes** (Grow from 4 to 16 to 48 to 256)
- Space per key is bounded (worst case 1 key per Node4)

TABLE I
SUMMARY OF THE NODE TYPES (16 BYTE HEADER, 64 BIT POINTERS).

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	$16 + 256 \cdot 8 = 2064$

TABLE II
WORST-CASE SPACE CONSUMPTION PER KEY (IN BYTES) FOR DIFFERENT RADIX TREE VARIANTS WITH 64 BIT POINTERS.

	$k = 32$	$k \to \infty$
ART	43	52
GPT	256	∞
LRT	2048	∞
KISS	>4096	NA.

What about **range queries**?

We sort the **keys**!
But in what order?

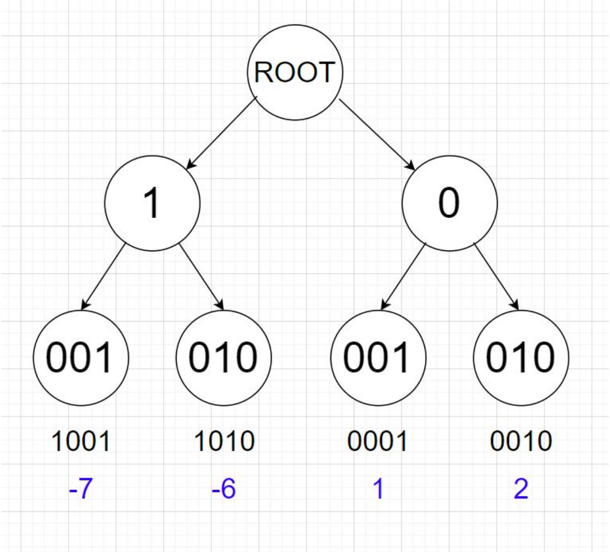
Binary-Comparable Keys

- Strings have lexicographic order (a<b<c)
- Signed integers have sign bit 0,1, where 1 is negative and 0 is positive, which is not lexicographically sorted
- Required transformations before storing in ART
- Sorted keys enabling efficient ordered range scans and lookups for minimum, maximum, top-N, etc

How to represent **-1** using 4 bit?

(0001 for **1**, 0100 for **4**)

Binary-Comparable Keys



Example: Two’s Complement

1 - 0001 or 0000 0001

-1 - 1111 or 1111 1111

For sign bit, 1 < 0

For example, to calculate the decimal number **−6** in binary from the number **6**:

- Step 1: +*6* in decimal is *0110* in binary; the leftmost significant bit (the first 0) is the sign *(just 110 in binary* would be -2 in decimal).
- Step 2: flip all bits in *0110*, giving *1001*.
- Step 3: add the place value 1 to the flipped number *1001*, giving *1010*.

Bits:	1	0	1	0
Decimal bit value:	−8	4	2	1
Binary calculation:	$-(1\times2^3)$	(0×2^2)	(1×2^1)	(0×2^0)
Decimal calculation:	−(**1**×8)	**0**	**1**×2	**0**

Binary-Comparable Keys

- Sorted keys makes data in sorted order.
- All operations that rely on this order can be supported
- Replace comparison-based trees with radix trees
- Replace comparison-based sorting algorithms like quicksort or mergesort with the radix sort algorithm

Evaluation

- Hardware Specifications

Component	Specification
CPU	Intel Core i7 3930K
Cores	6
Threads	12
Clock Rate	3.2 GHz
Turbo Frequency	3.8 GHz
Cache	12 MB shared, last-level
RAM	32 GB quad-channel DDR3-1600
Operating System	Linux 3.2 (64 bit)
Compiler	GCC 4.6

Contestants

Adaptive Radix Tree (ART)

- Generalized Prefix Tree (GPT) —— radix tree
- Cache-Sensitive B+-tree (CSB) —— optimized for main memory
- k-ary Search Tree (kary) —— read-only
- Fast Architecture Sensitive Tree (FAST) —— read-only
- Hash Table (HT)
- Red-Black Tree (RB)

Evaluation

- Search Performance
- Caching Effects
- Updates
- End-to-End Evaluation

Search Performance

Similar -> Cache effects

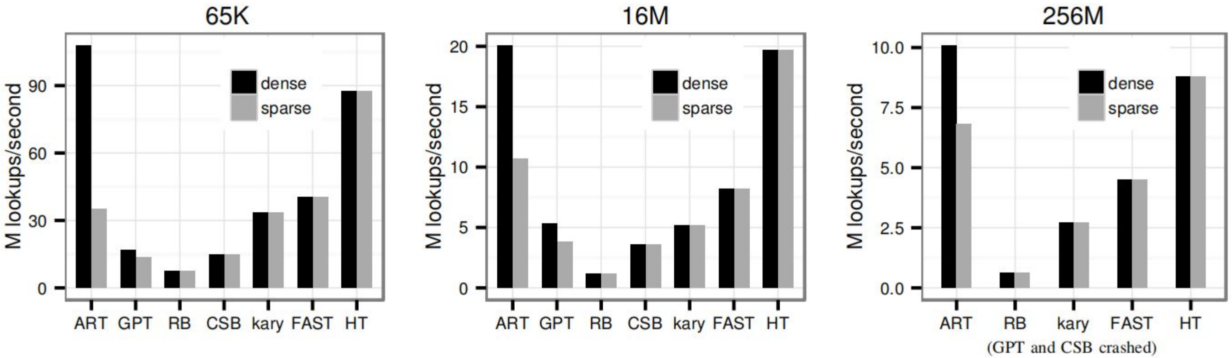


Fig. 10. Single-threaded lookup throughput in an index with 65K, 16M, and 256M keys.

Search Performance

TABLE III
PERFORMANCE COUNTERS PER LOOKUP.

	65K			16M		
	ART (d./s.)	FAST	HT	ART (d./s.)	FAST	HT
Cycles	40/105	94	44	188/352	461	191
Instructions	85/127	75	26	88/99	110	26
Misp. Branches	0.0/0.85	0.0	0.26	0.0/0.84	0.0	0.25
L3 Hits	0.65/1.9	4.7	2.2	2.6/3.0	2.5	2.1
L3 Misses	0.0/0.0	0.0	0.0	1.2/2.6	2.4	2.4

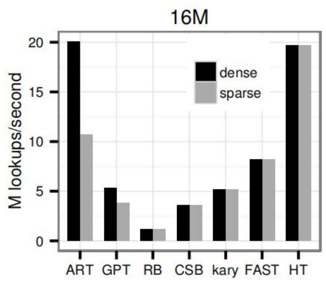
- Cycles: Processor cycles taken per lookup operation;
  a. fewer cycles indicate higher efficiency.
- Instructions: Number of instructions executed per lookup;
  a. fewer instructions suggest a more efficient algorithm.
- Mispredicted Branches: Counts the times the processor's branch prediction is wrong;
  a. fewer mispredictions lead to better performance.
- L3 Cache Hits: How often the searched data is found in the L3 cache during lookups;
  a. more hits typically mean better performance.
- L3 Cache Misses: Instances where the data is not found in the L3 cache, indicating a need to access slower main memory;
  a. fewer misses are ideal.

Search Performance

From one query, one thread,

to using multiple unsynchronized threads

Interleave multiple tree traversals using **software pipelining**



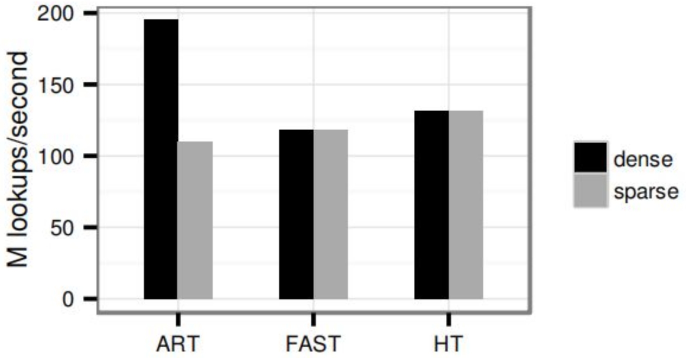


Fig. 11. Multi-threaded lookup throughput in an index with 16M keys (12 threads, software pipelining with 8 queries per thread).

Caching Effects

DRAM (dynamic random access memory)

DRAM latency amounts to hundreds of CPU cycles in today’s CPU

- skew
- size

Cache Effects – skew

Skew: the imbalance in the frequency of access or distribution of the keys in a dataset

Cache misses decreases as the skew increases

Fast requires more comparisons and offset calculations

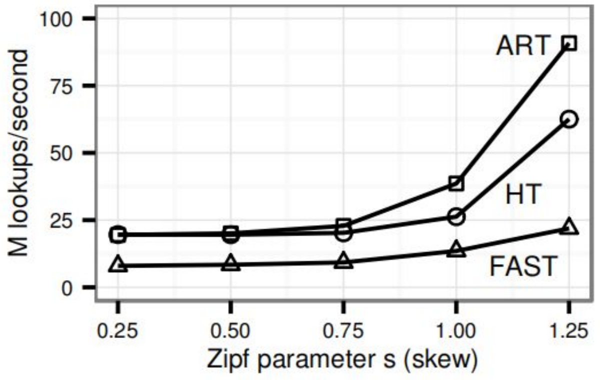


Fig. 12. Impact of skew on search performance (16M keys).

Cache Effects – cache size

Consider competing memory accesses

HT is mostly unaffected

ART can adapt flexibly

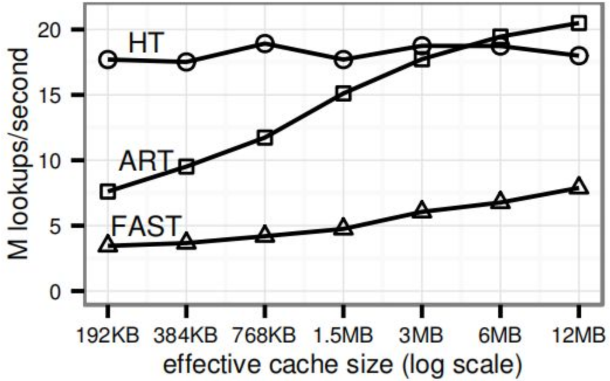


Fig. 13. Impact of cache size on search performance (16M keys).

Updates

For ART, trade off between

1. Time consuming for data structure adaptations
2. Time saving from the space saving

Ordered data benefits ordered search method, (only HT excluded)

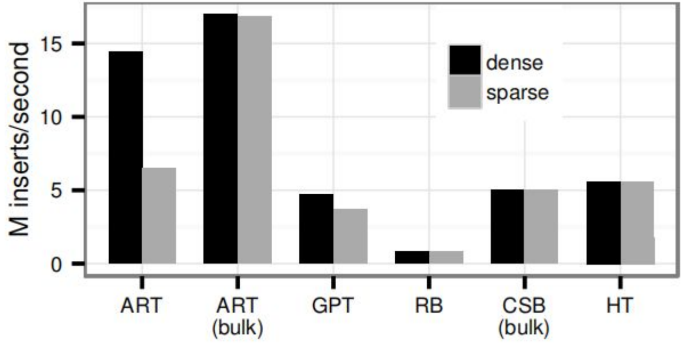


Fig. 14. Insertion of 16M keys into an empty index structure.

Updates

delta mechanism

Merge FAST and red-black tree periodically

O(n) merging step

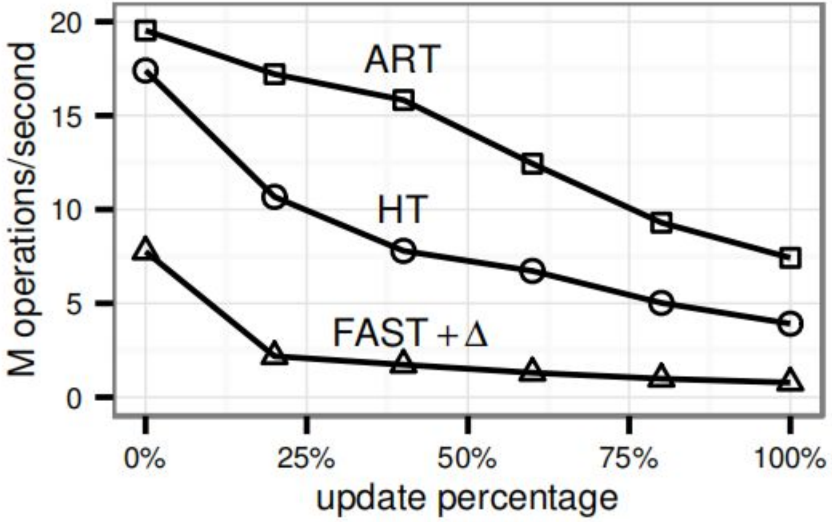


Fig. 15. Mix of lookups, insertions, and deletions (16M keys).

End-to-End Evaluation

System: HyPer

Compare different data structures

Implement diverse related operations

uses TPC-C, a standard OLTP benchmark

HyPer

- supports both transactional (OLTP) and analytical (OLAP) workloads
- Performance relies critically on indexes（no overhead for buffer management, locking, or latching）

- OLTP: Online Transaction Processing
- OLAP: Online Analytical Processing

End-to-End Evaluation

System: HyPer

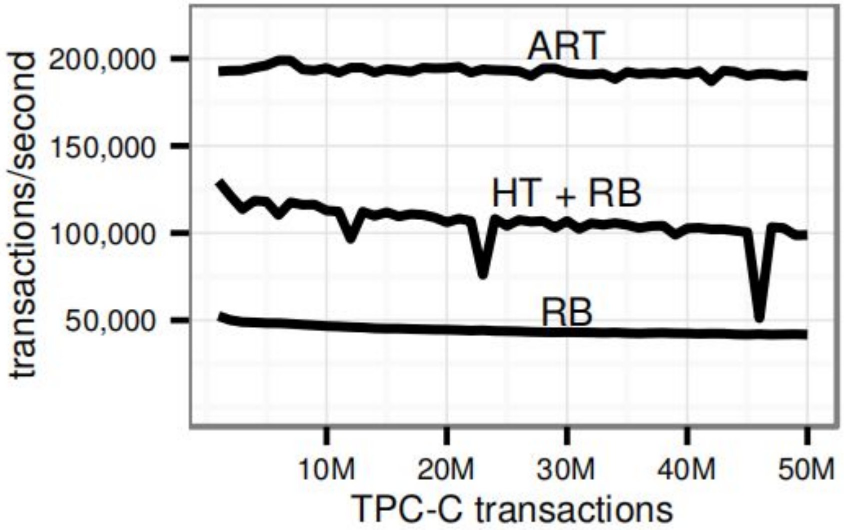
Compare different data structures

Implement diverse related operations (read, range scan, prefix lookup, minimum, etc)

uses TPC-C, a standard OLTP benchmark

TPC-P transactions

TPC-C requires prefix-based range scans for some indexes, so cannot use hash tables for all indexes.



Space Consumption

Index 3: relatively long strings

Indexes 1, 2, 4, and 5: dense integers

MAJOR TPC-C INDEXES AND SPACE CONSUMPTION PER KEY USING ART.

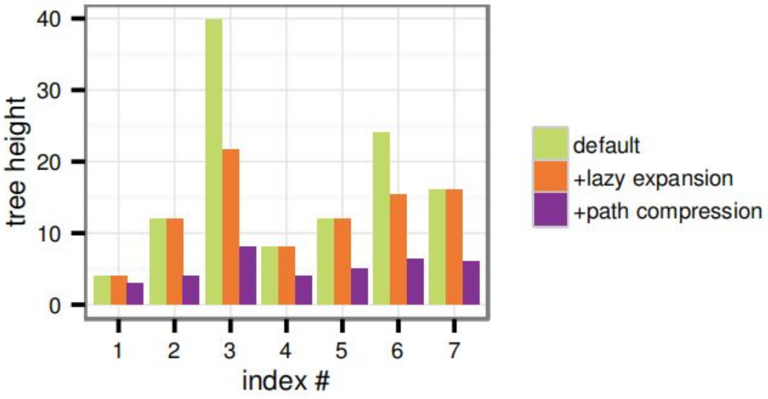
#	Relation	Cardinality	Attribute Types	Space
1	item	100,000	int	8.1
2	customer	150,000	int,int,int	8.3
3	customer	150,000	int,int,varchar(16),varchar(16),TID	32.6
4	stock	500,000	int,int	8.1
5	order	22,177,650	int,int,int	8.1
6	order	22,177,650	int,int,int,int,TID	24.9
7	orderline	221,712,415	int,int,int,int	16.8

Final Results

Lazy expansion helps with index 3, 6, which have TID

- TID leads to sparser distribution of keys
- With the lazy expansion, TID can be truncated

Path compression helps with all indexes



Conclusions

Lazy expansion and path compression

ART is faster than most of state-of-the-art main-memory data structures.

ART is faster than the read-only FAST.

Only Hash Table is competitive, but HT is unsorted.

Future works

- synchronizing concurrent updates
- design a space-efficient radix tree which has nodes of equal size

Thanks