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

Ivan Sinyagin

Memory Wall

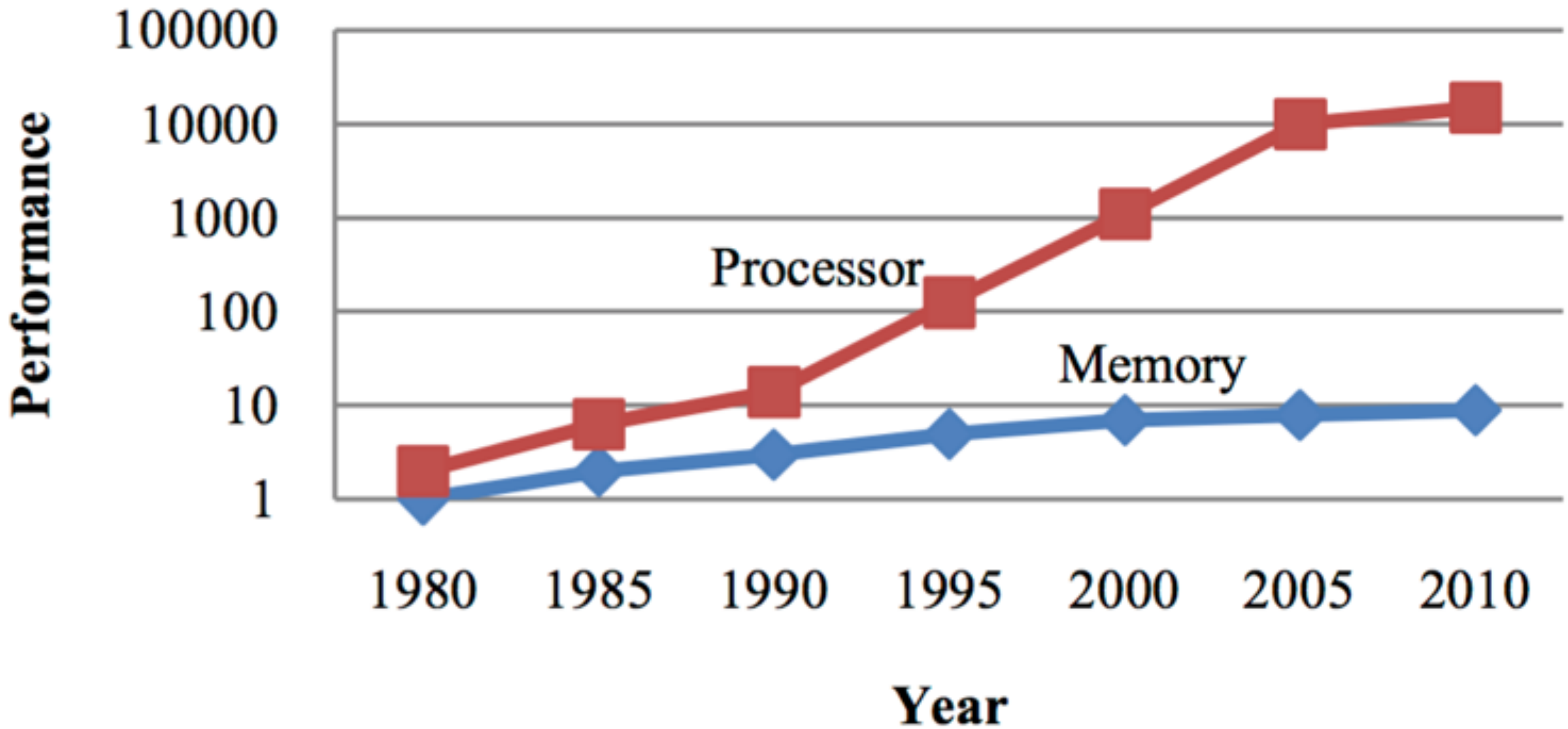


Figure 1. Processor - Memory Performance Imbalance [2]

Why indexes ?

Best data structure

$O(1)$?

Binary Search !

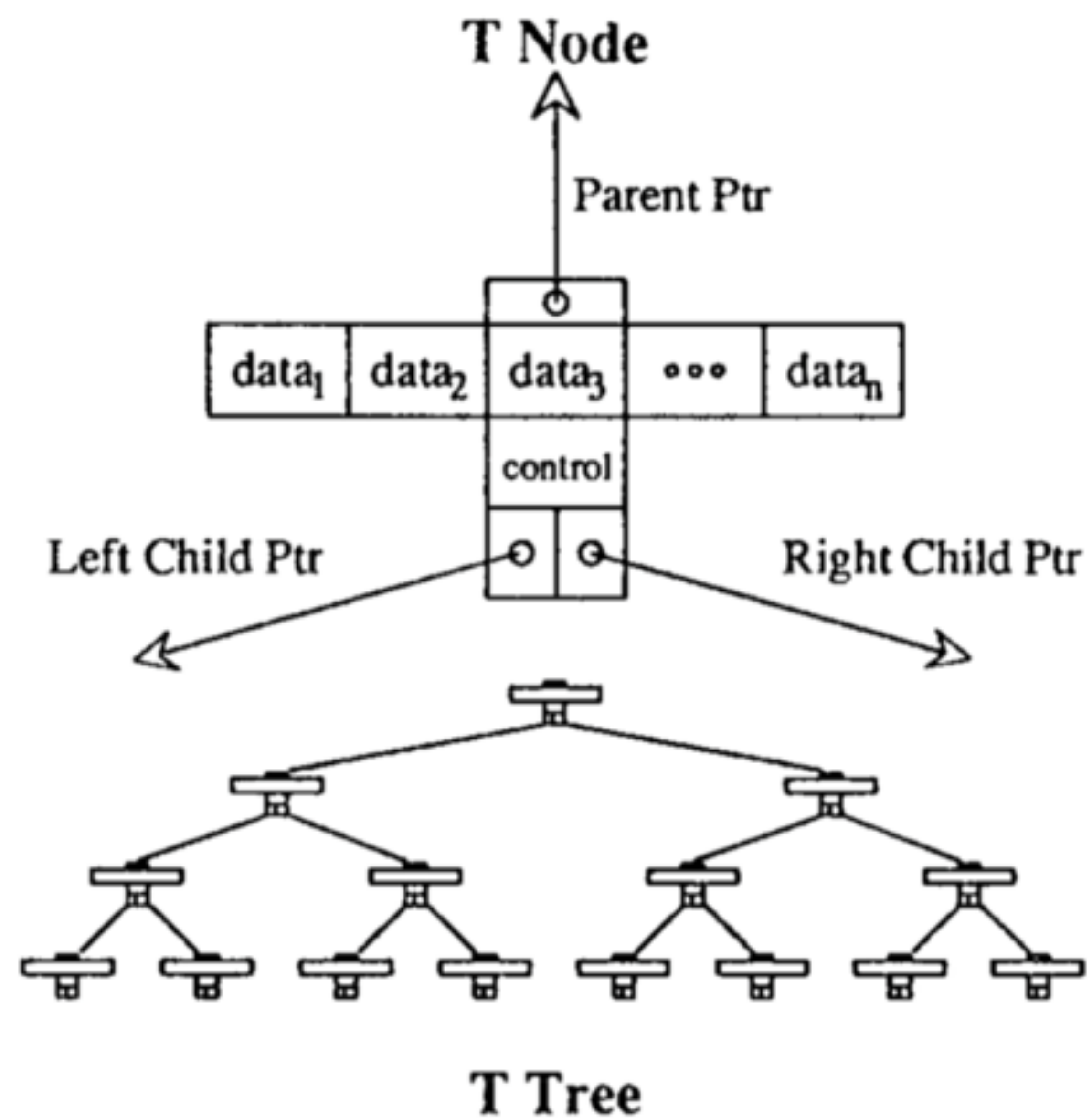
Binary Search

- 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

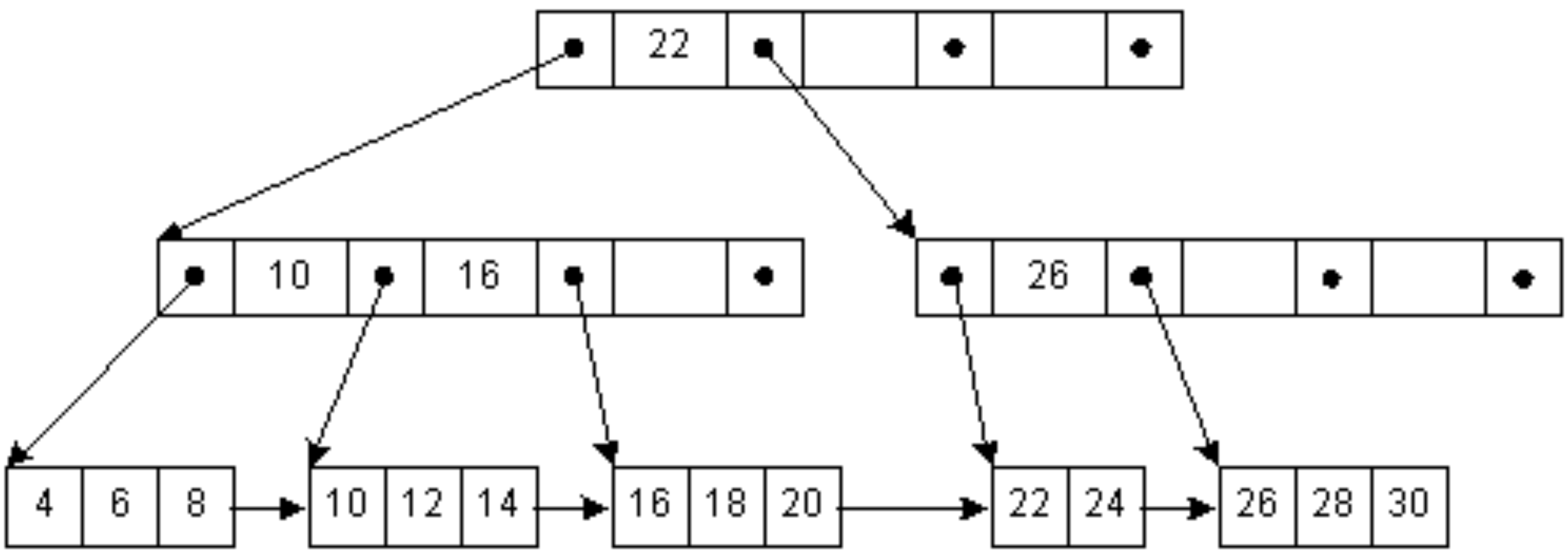
Trees

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



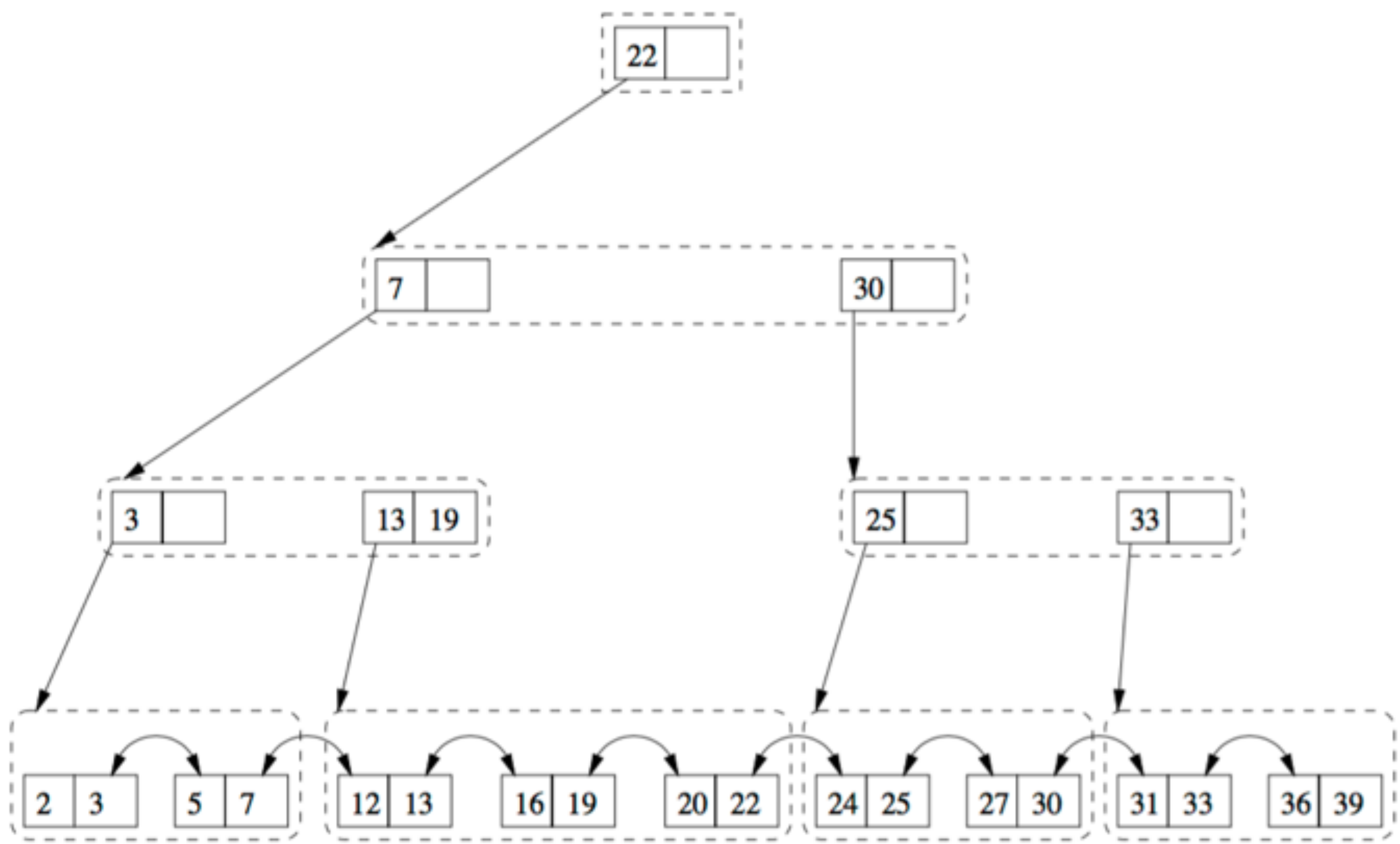
B+ tree



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



CSB+ tree

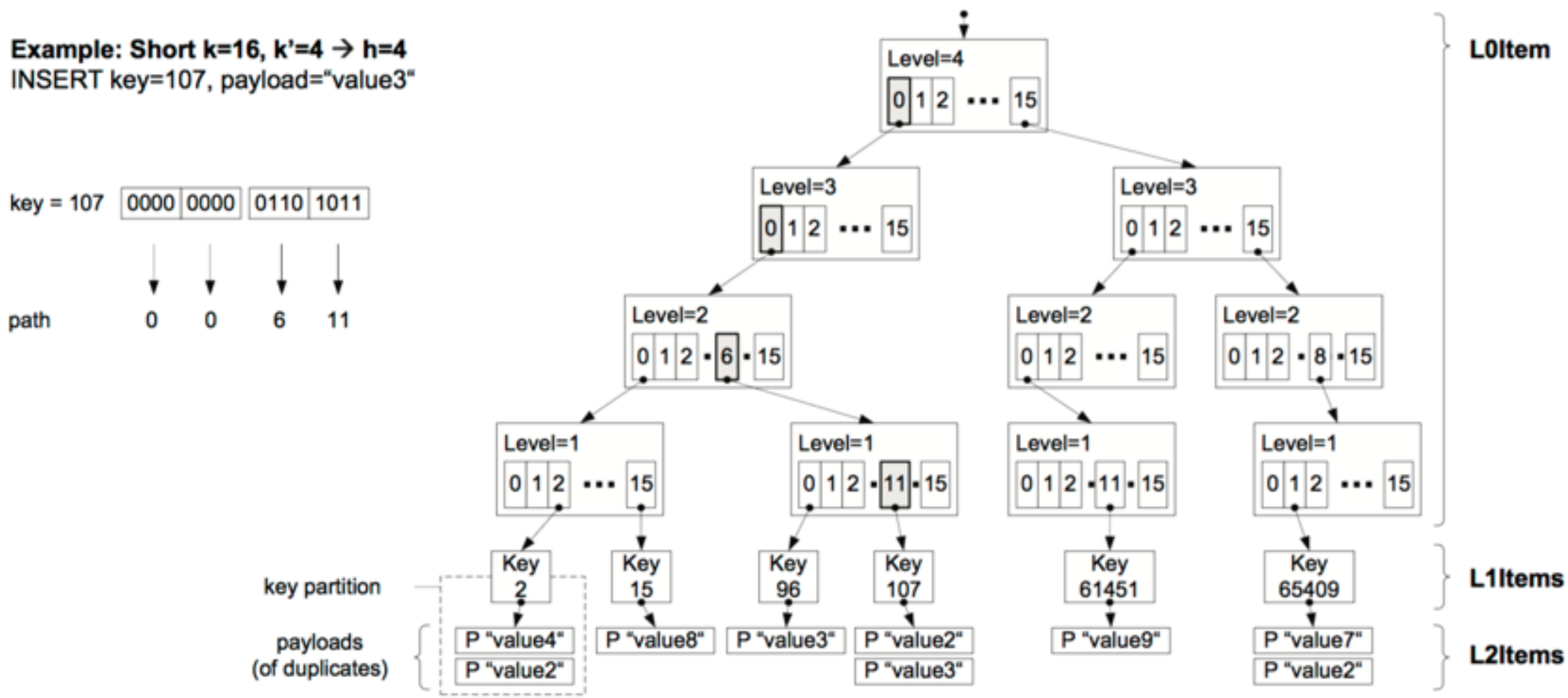
- ~ 1999-2000
- Improved space complexity
- Great cache line utilization: keys + 1 pointer
- Node size ~ cache line
- Update overhead - more logic to balance

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



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

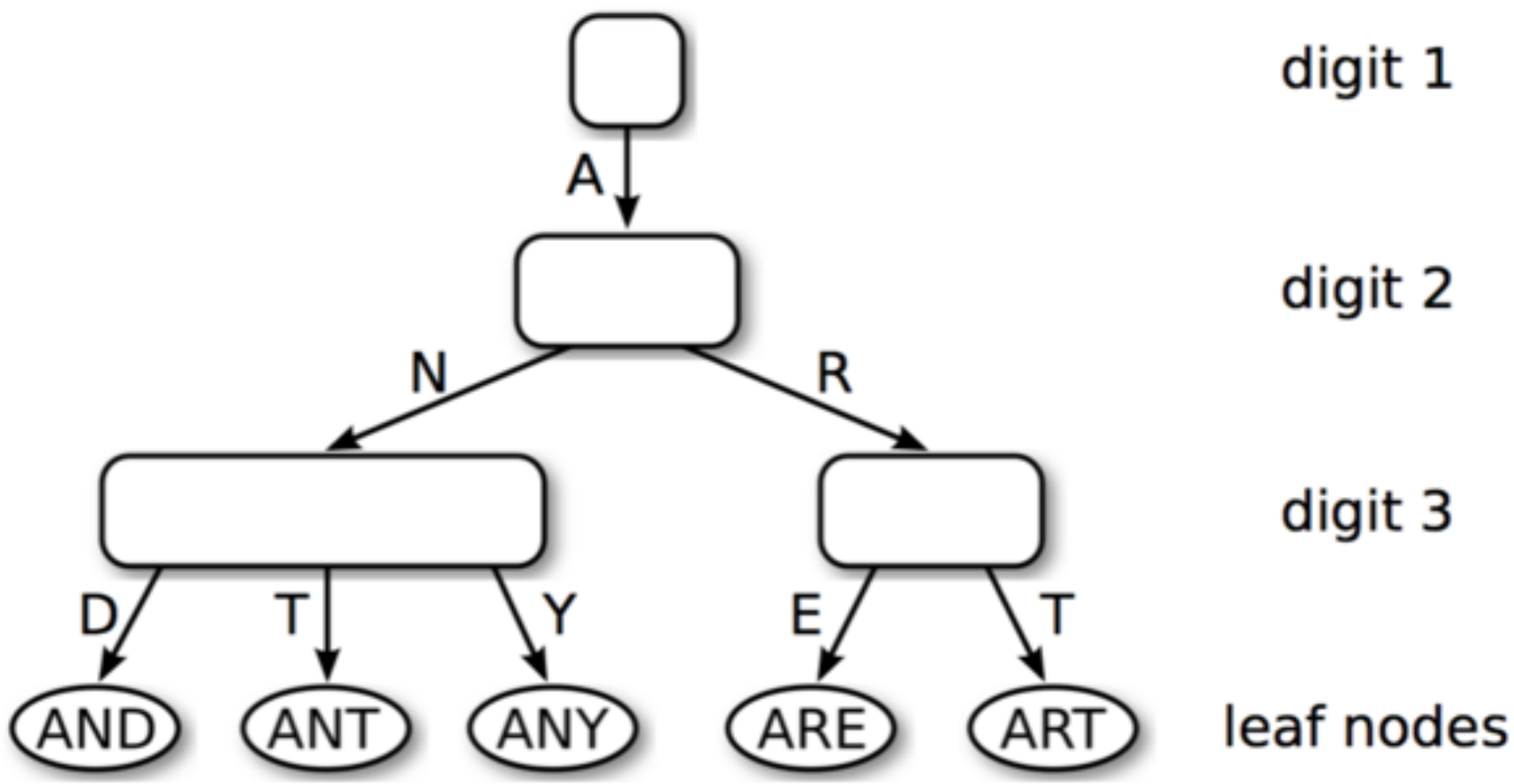
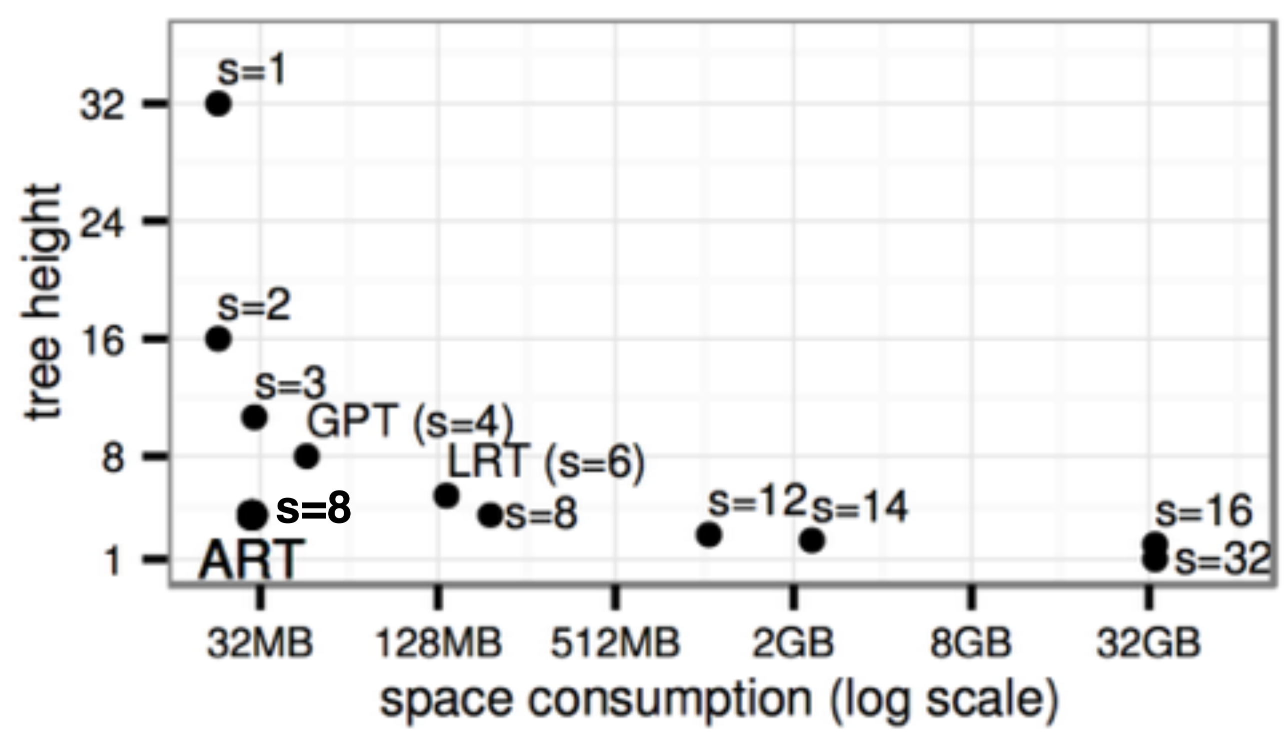


Fig. 1. Adaptively sized nodes in our radix tree.

ART height

- 1M keys
- ART height ~ B+ tree



Adaptive nodes

N256 implicit keys

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

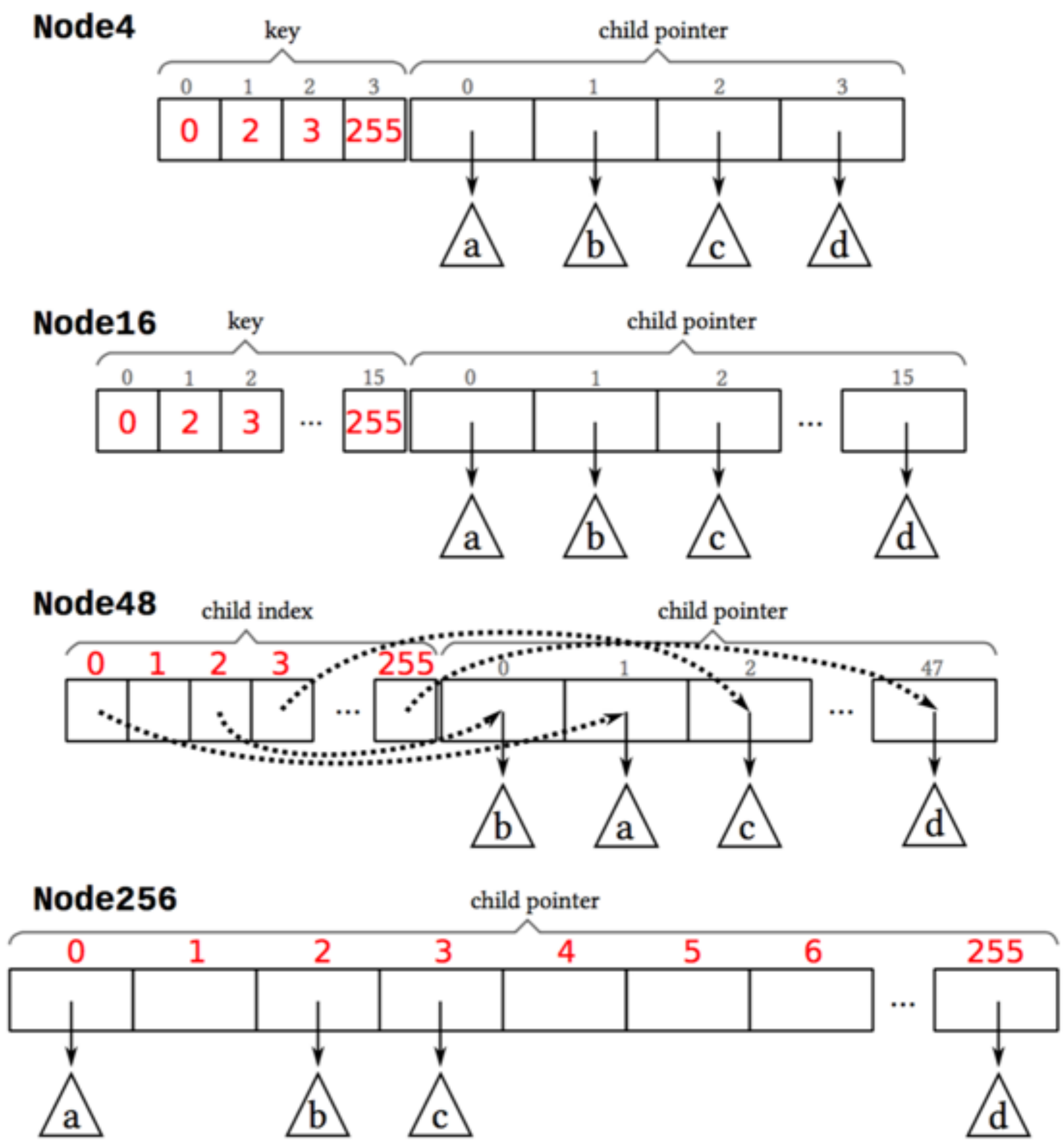
N4 & N16 explicit keys

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

Why 16?

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

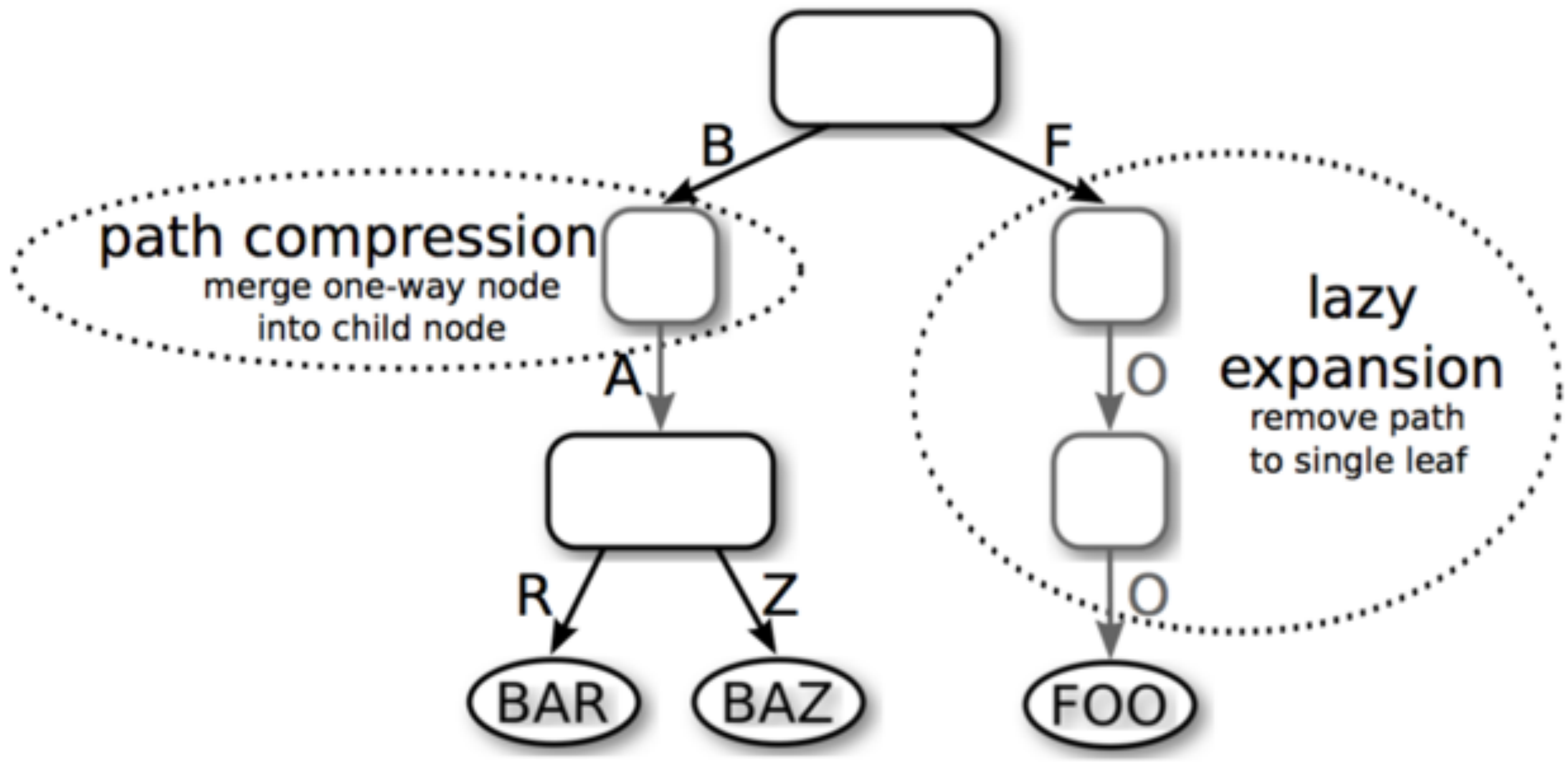
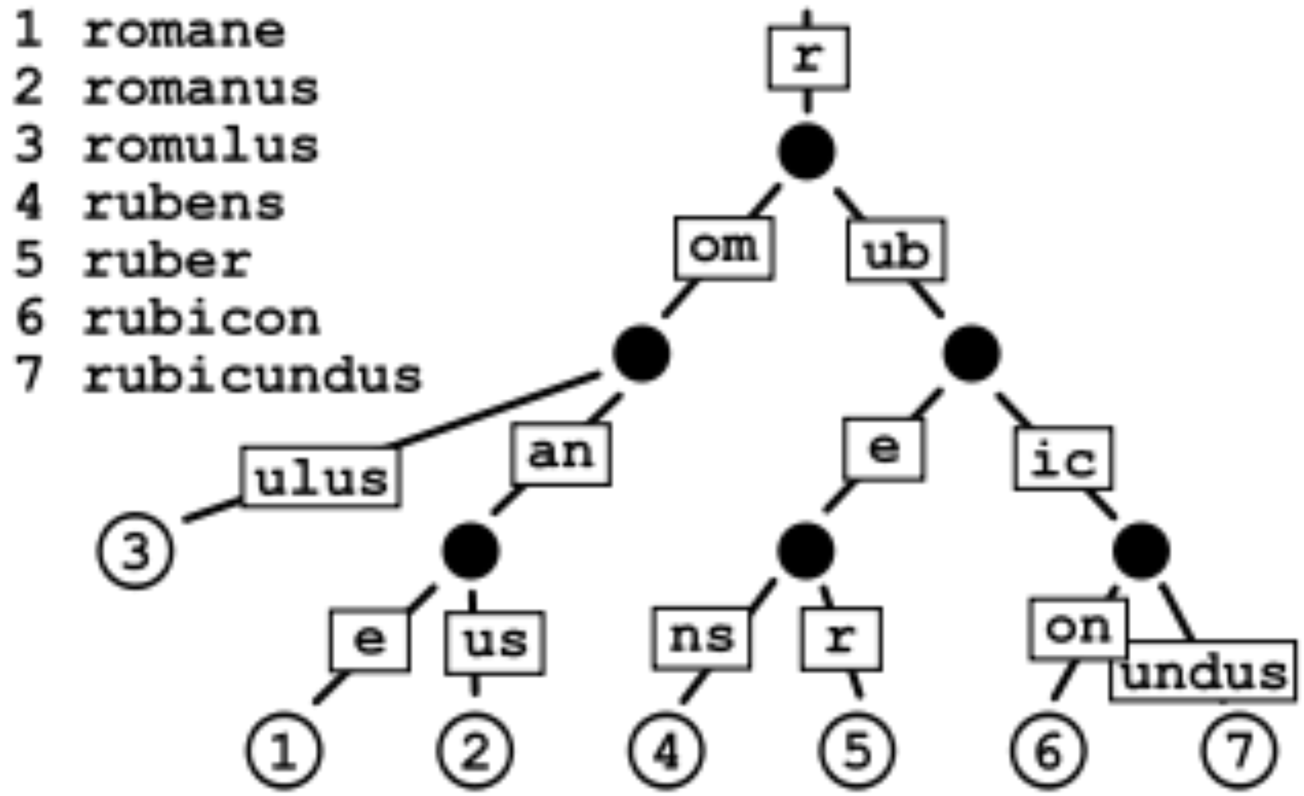


Fig. 6. Illustration of lazy expansion and path compression.



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

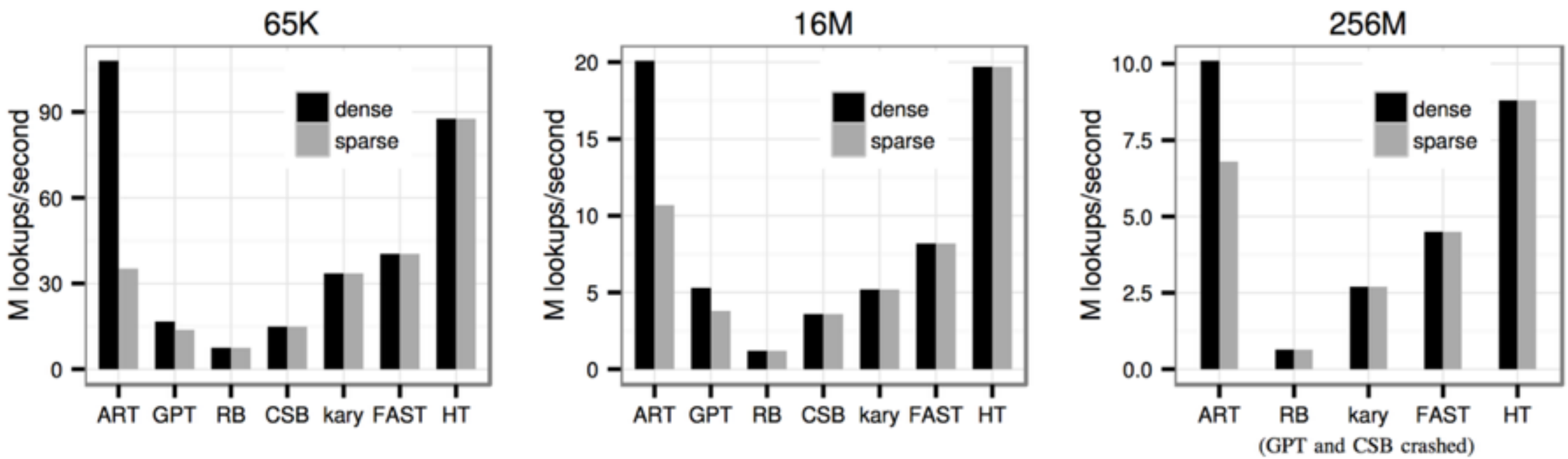


Fig. 10. Single-threaded lookup throughput in an index with 65K, 16M, and 256M keys.

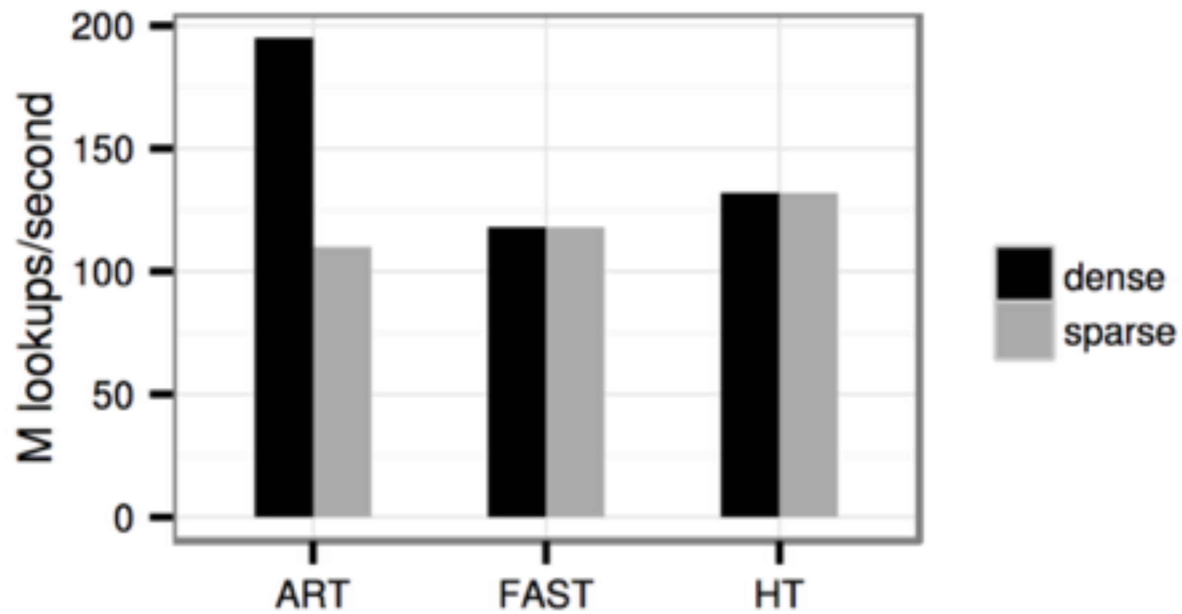
Mispredictions and Misses

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

- 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

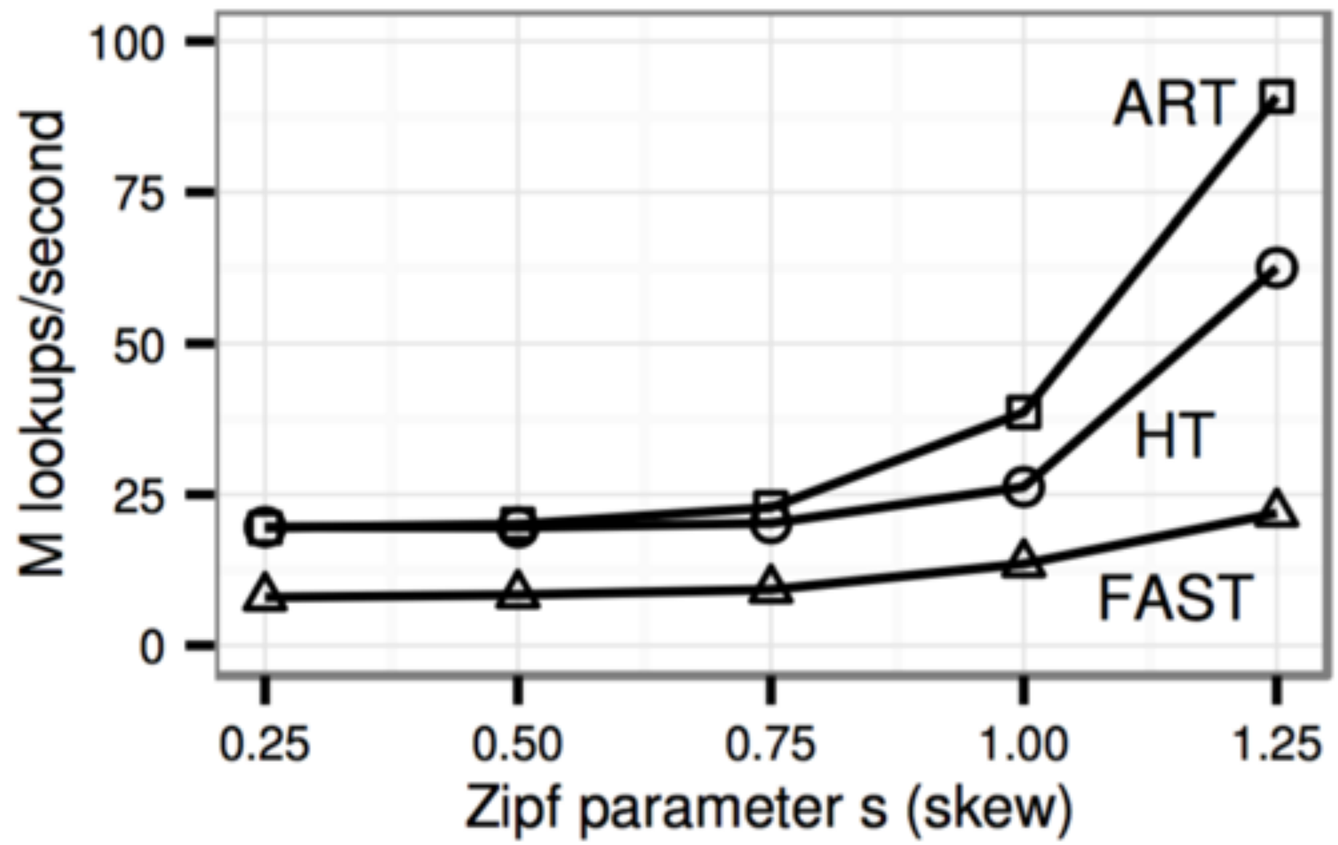


Fig. 12. Impact of skew on search performance (16M keys).

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

Round-robin dense search: cache size

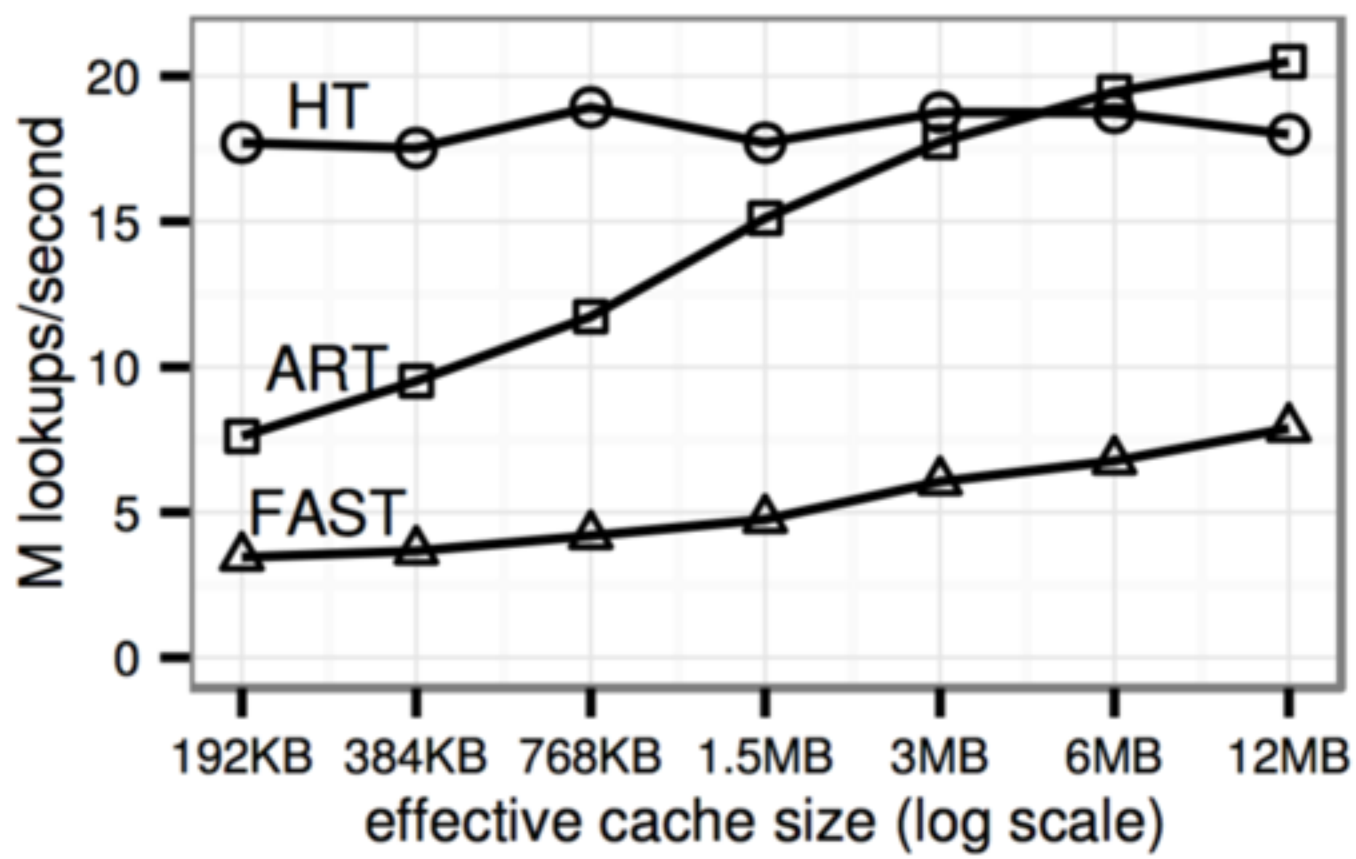


Fig. 13. Impact of cache size on search performance (16M keys).

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

Inserts

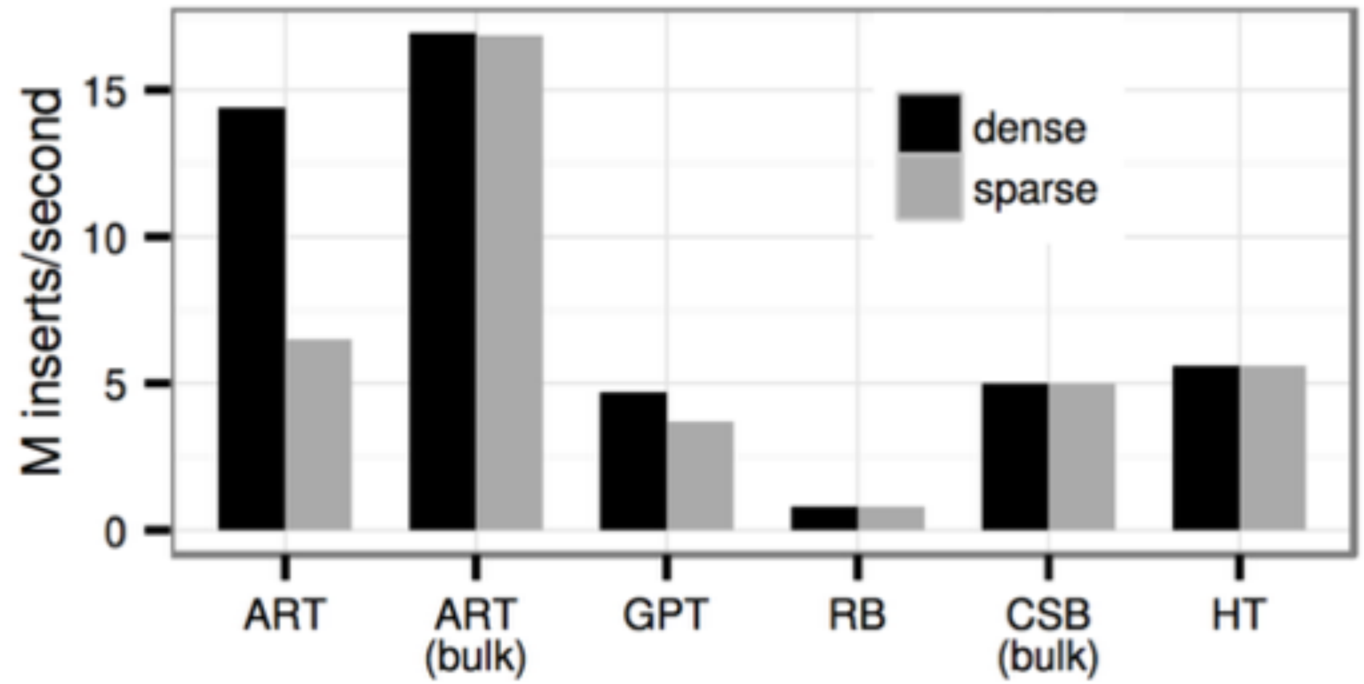


Fig. 14. Insertion of 16M keys into an empty index structure.

- 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

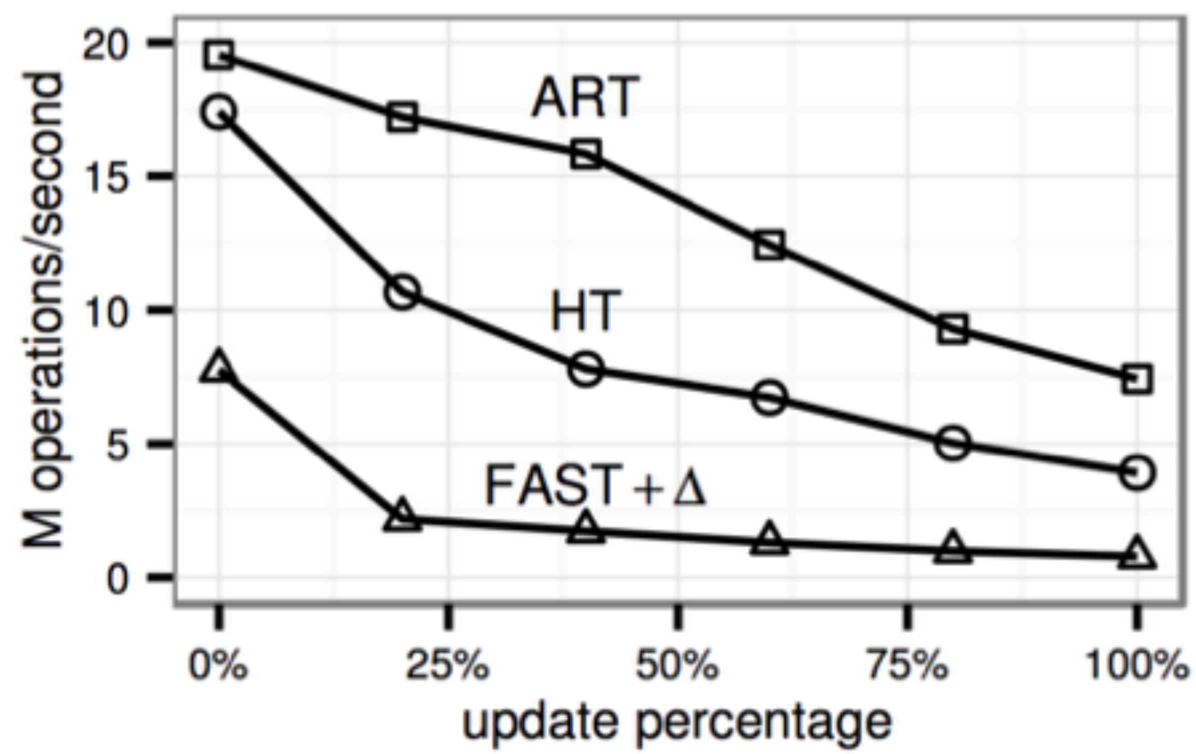


Fig. 15. Mix of lookups, insertions, and deletions (16M keys).

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

HyPer OLTP

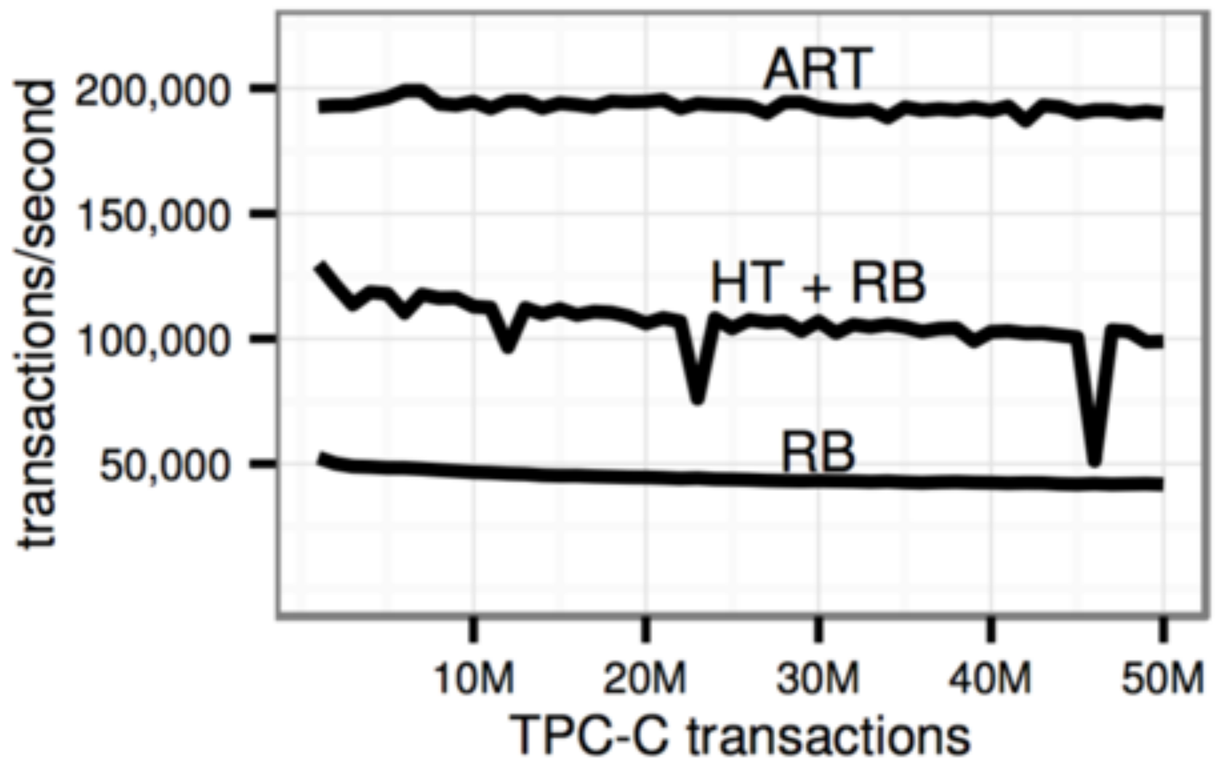
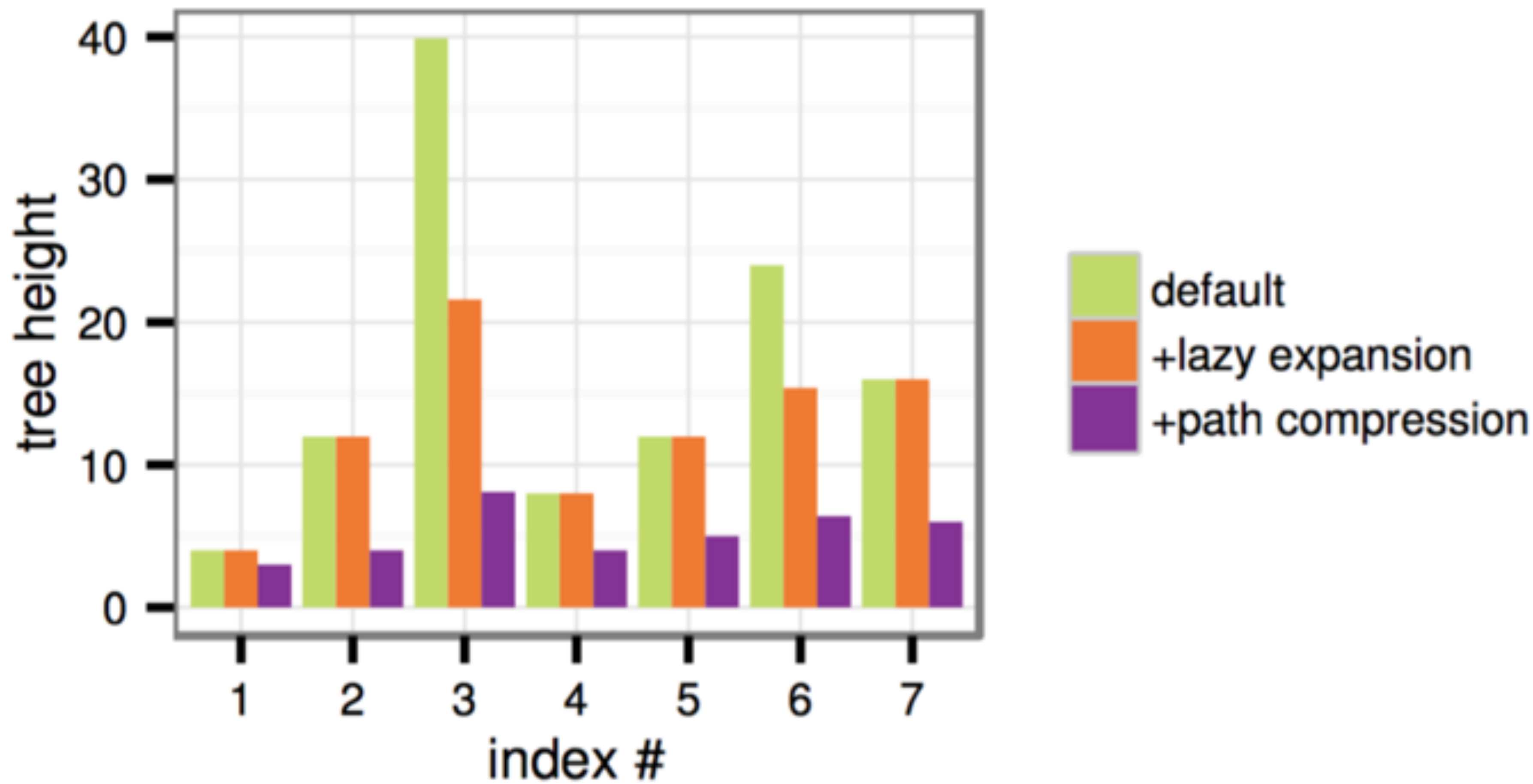


Fig. 16. TPC-C performance.

- 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

TABLE IV
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

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

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