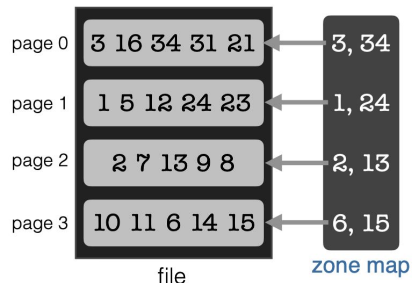
Adaptive Radix Tree Indexes

By Tal Kronrod

ARTful Indexing for Main-Memory Databases
Viktor Leis, Alfons Kemper, Thomas Neumann

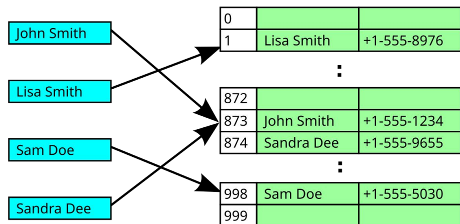
Why Do We Need Indexes?

Facilitate faster queries!



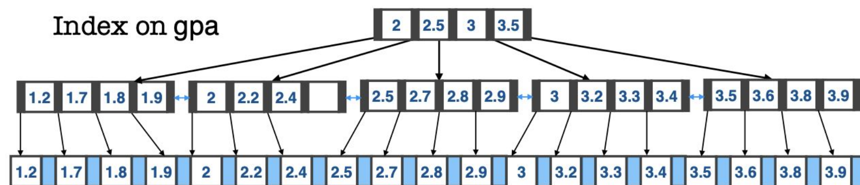
What are some examples of indexes?

What are the main properties of indexes?



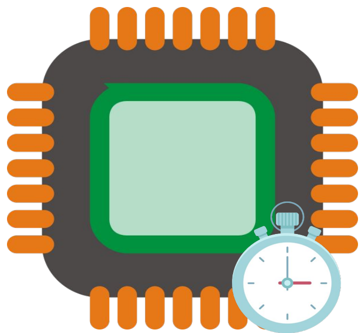
Small and fit in memory

Help avoid disk I/O



Current State of Database Systems (2013)

Most databases can fit into memory!



CPUs bound performance

Indexes are a CPU bottleneck

If the data fits in memory, do we even need indexes? Why?

Cache Misses Are Still Expensive!

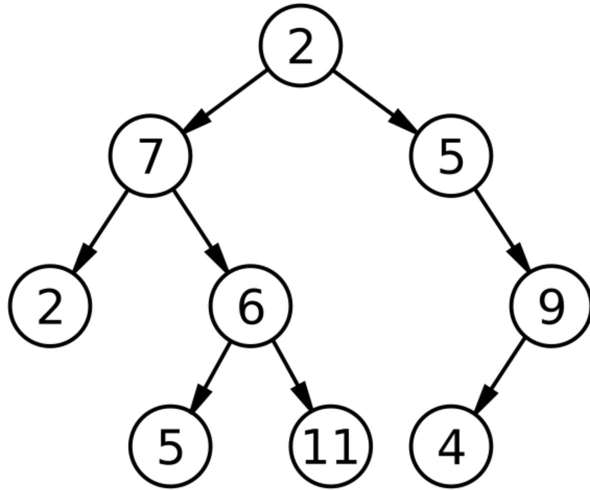
Indexes reduce cache misses

Better cache utilization = higher throughput

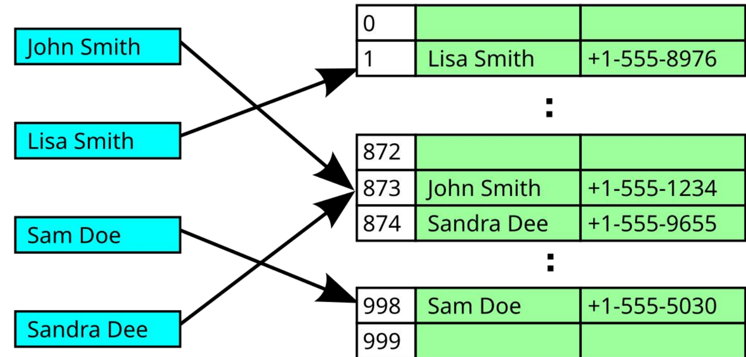
How fast is access?

Access latency	Memory type	Scaling up
1 ns	CPU/register	1 s
4 ns	on-chip cache	4 s
10 ns	on-board cache	10 s
100 ns	DRAM	100 s
16,000 ns	SSD	4.44 hours
2,000,000 ns	HDD	3.3 weeks
1,000,000,000 ns	Tape	31.7 years

Current State-of-the-Art



Tree Based Indexes



Hash Tables

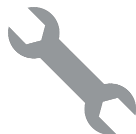
What are some limitations of these structures?

Do we have a problem?

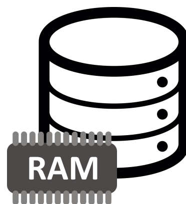
Yes!

In that case: what should our ideal data structure be able to do?

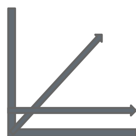
1. Index structure that supports all primitive operations



2. Well tuned to modern hardware



3. Works well for in-memory databases

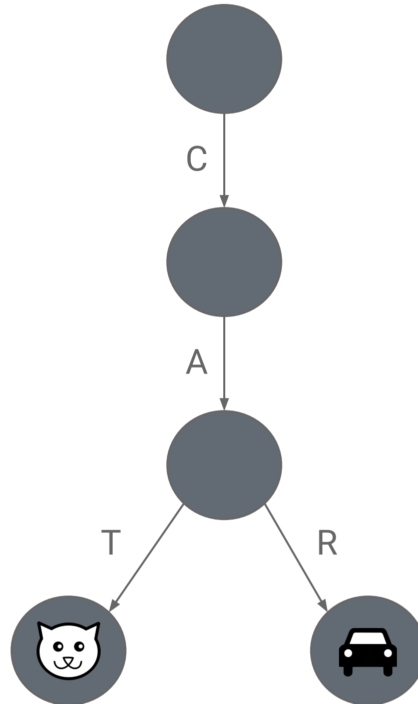


4. Separate worst-case performance from the size of the database

Radix Trees

What is a Radix Tree?

Path to the lead node represents the key



Span (s) = fanout in B+ Tree

What do primitive operations look like?

Insert CAT

Insert CAR

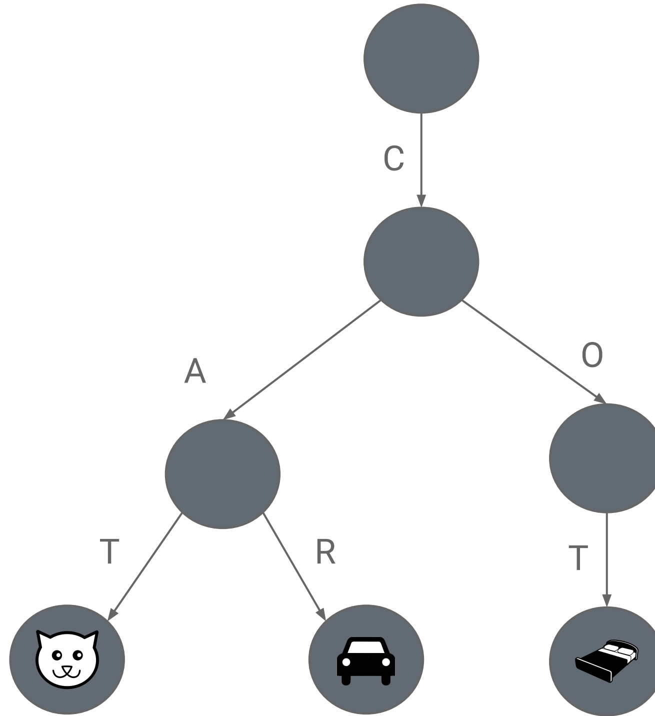
Insert COT

PQ CAR

PQ CON

RQ CAA-CAZ

Delete CAT



Radix: A Separation of Worst-Case Performance

Operations are bounded by the length of the longest key rather than the size of the database!

As our database grows, we will retain consistent performance (unlike other index structures).

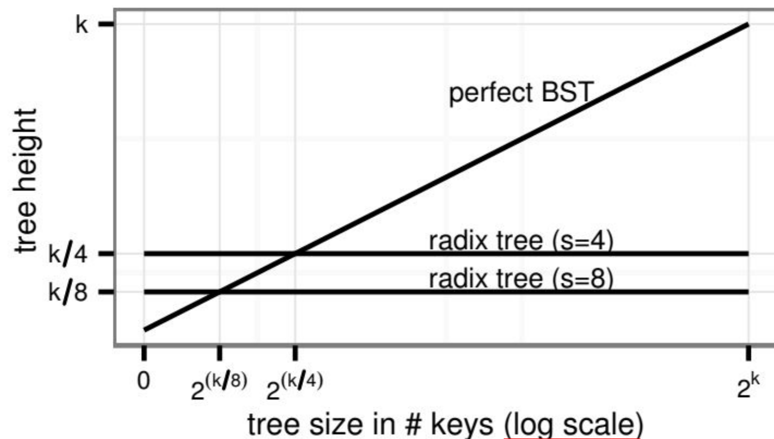


Fig. 2. Tree height of perfectly balanced binary search trees and radix trees.

s = Node span
k = Key length

What are the Limitations of Radix Trees

1. Inefficient for sparse data sets

2. Work poorly for long keys

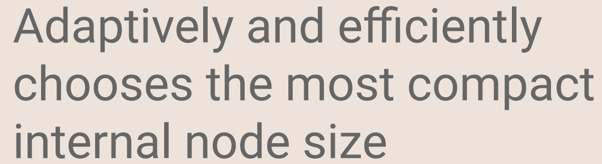
3. Unbounded space consumption

Adaptive Radix Trees (ART)

Features Radix Trees Should Have

1. Adaptively and efficiently chooses the most compact internal node size
2. Collapse unnecessary inner nodes
3. Worst case space consumption is bounded

Adaptive Inner Nodes



Balancing Span Size with Space Consumption

- We want to have a large span for faster lookups
- Large spans lead to a lot of unused space in intermediate nodes

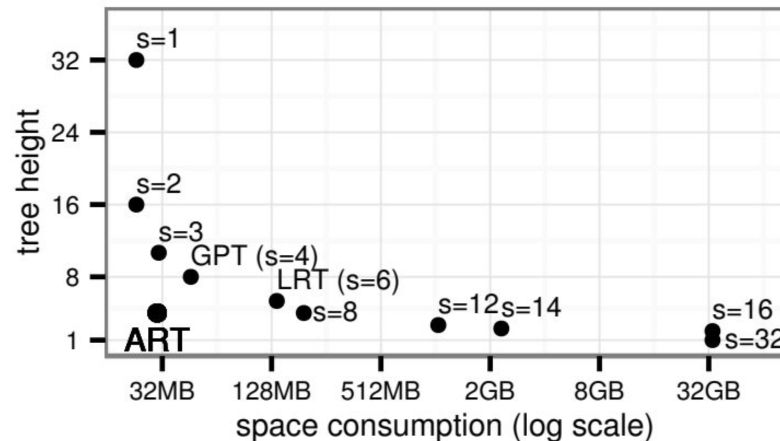


Fig. 3. Tree height and space consumption for different values of the span parameter s when storing 1M uniformly distributed 32 bit integers. Pointers are 8 byte long and nodes are expanded lazily.

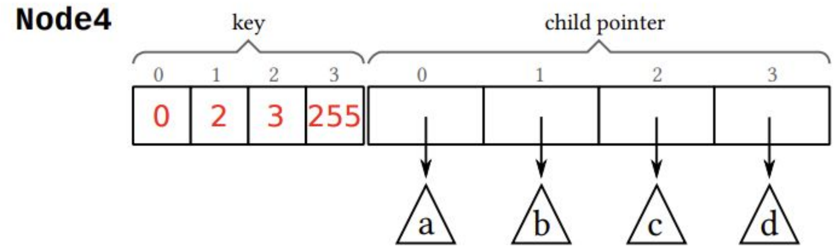
How would you balance these two?

Resizeable Nodes!

- Naive approach: resize on every insert/delete
- Actually: have 4 fixed-sized nodes
- Use a span of 8 bits
  - Large fanout
  - Simplifies implementation
- 16 bit headers that store:
  - Node type
  - Number of children
  - Compressed path

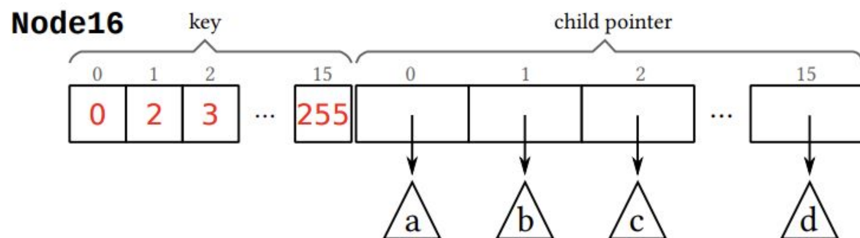
Node4

- Stores up to 4 child pointers
- Keys are sorted
- Values are stored in corresponding index of the key



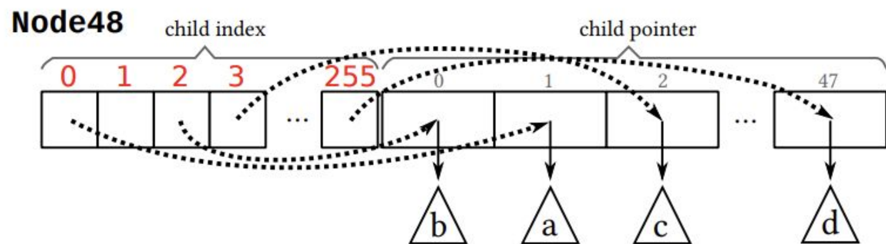
Node16

- Stores between 5 - 16 children
- Same storage layout as Node4
- Can find keys using binary search or parallel SIMD comparisons on modern hardware



Node48

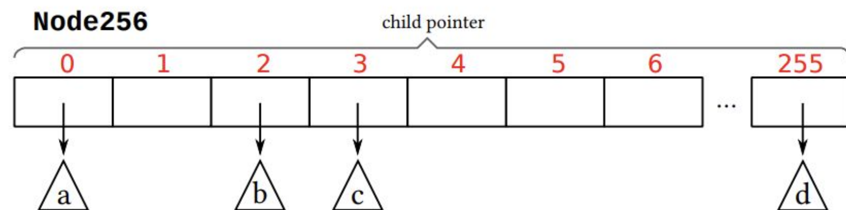
- Keys are stored in a 256 byte map that stores the index in the value array
- Value array is 48 elements long



Is there a reason this node needs to be capped at 48?

Node256

- 256 byte array that directly stores the pointers to the subtrees
- A map!



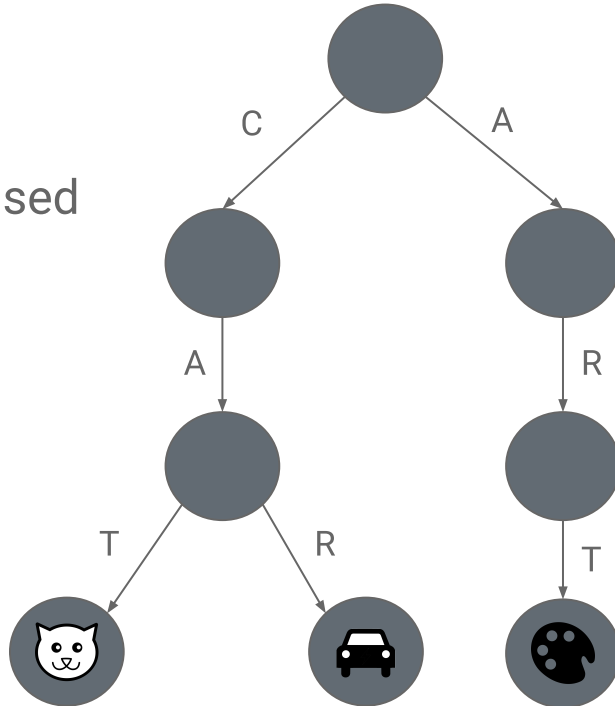
Options for Leaf Nodes

- Single-value leaves
- Multi-value leaves: Same as inner nodes, but store values rather than subtrees
- Combined pointer/value slots: inner nodes can store both subtrees and leaf values

Tree Compaction

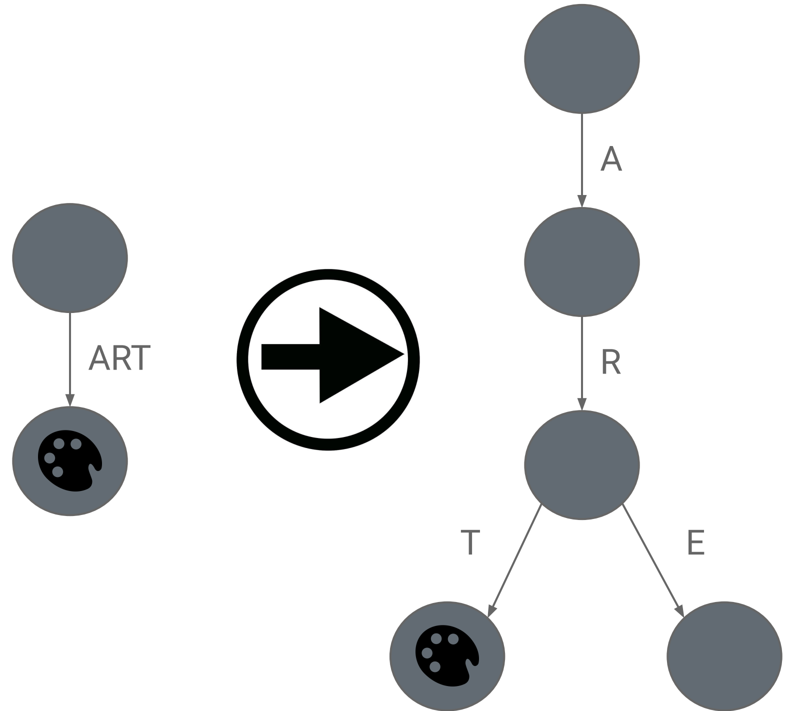
Consider this Radix Tree:

Is space being used efficiently?



Lazy Expansion

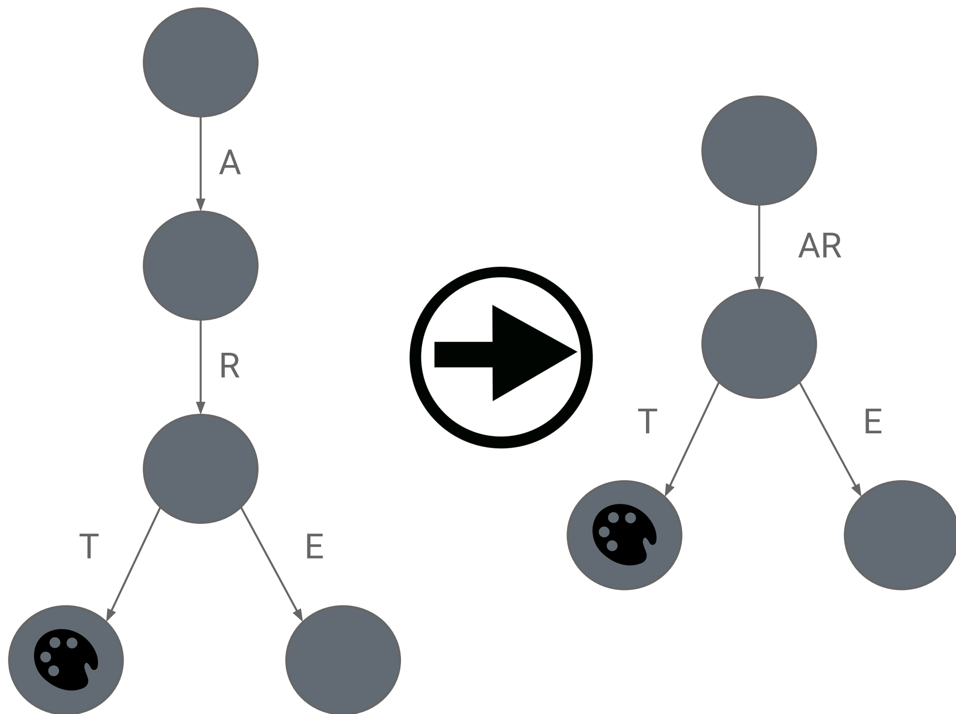
- Inner nodes are only created when they need to distinguish two or more leaf nodes



Path Compression

- removes all inner nodes that have only a single child

How would you deal with the removed key nodes?



Pessimistic Approach

- Store a partial key vector
- Stores the keys of all previous removed inner nodes

- Compare search key at all levels

Optimistic Approach

- Store the count of removed previous inner nodes
- Equal to the length of the vector

- Compare search key when a “wrong turn” was taken

What are the pros and cons of these approaches?

Bounded Space Consumption

Worst-Case Space Consumption Per Node

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	$16 + 256 \cdot 8 = 2064$

What is the worst case scenario?

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
General Prefix Tree	GPT	256	∞
Linux Kernel Radix Tree	LRT	2048	∞
	KISS	>4096	NA.

Experimental Results

Experimental Setup

- Intel Core i7 3930K CPU
  - 6 cores, 12 threads, 3.2 GHz, 3.8 GHz turbo frequency
- 12MB shared, last-level cache
- 32GB quad-channel DDR3-1600 RAM
- 64bit Linux 3.2

- To execute TPC-C:
  - Integrated ART into HyPer

The Competition

- Cache sensitive B+ tree (CSB)
  - A B+ tree optimized for main-memory
- K-Ary Search Tree & Fast Architecture Sensitive Tree (FAST)
  - Read-only search indexes optimized for modern x86 CPUs
- General Prefix Tree (GPT)
- Classic Red-Black Tree (RB)
- Chained hash table (HT) using MurmurHash64A for 64-bit platforms

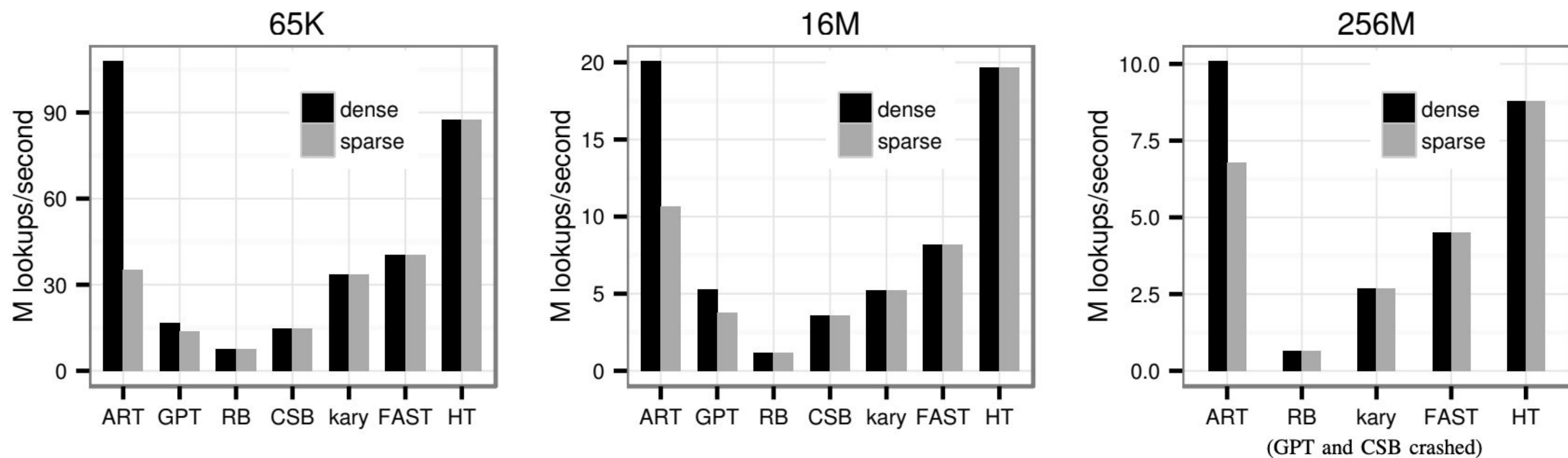


Fig. 10. Single-threaded lookup throughput in an index with 65K, 16M, and 256M keys.

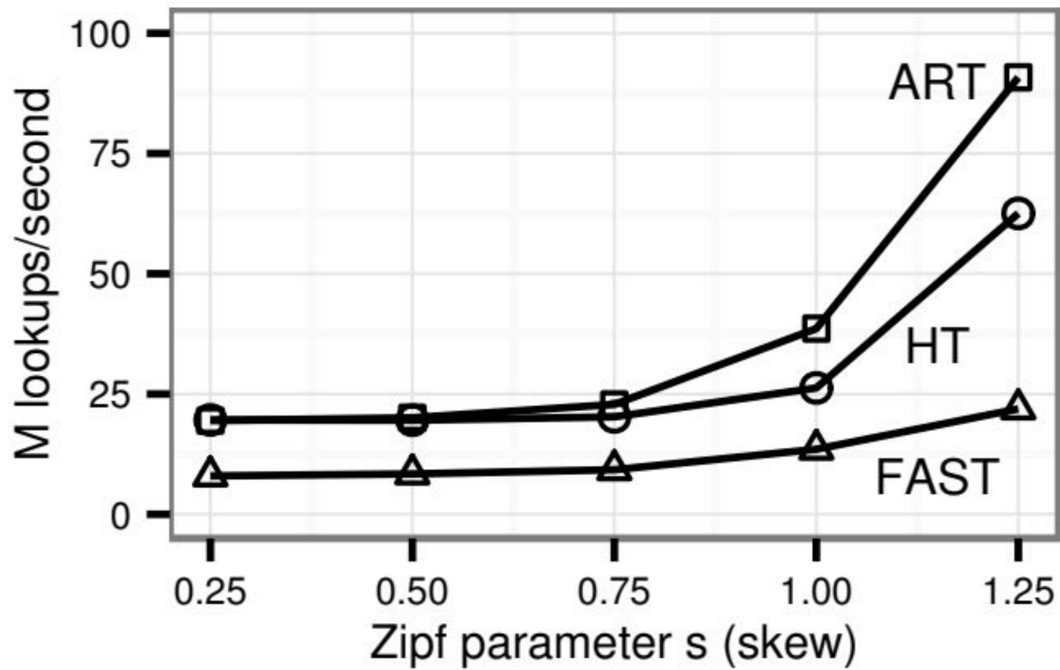
Microbenchmarks - Search Performance

TABLE III

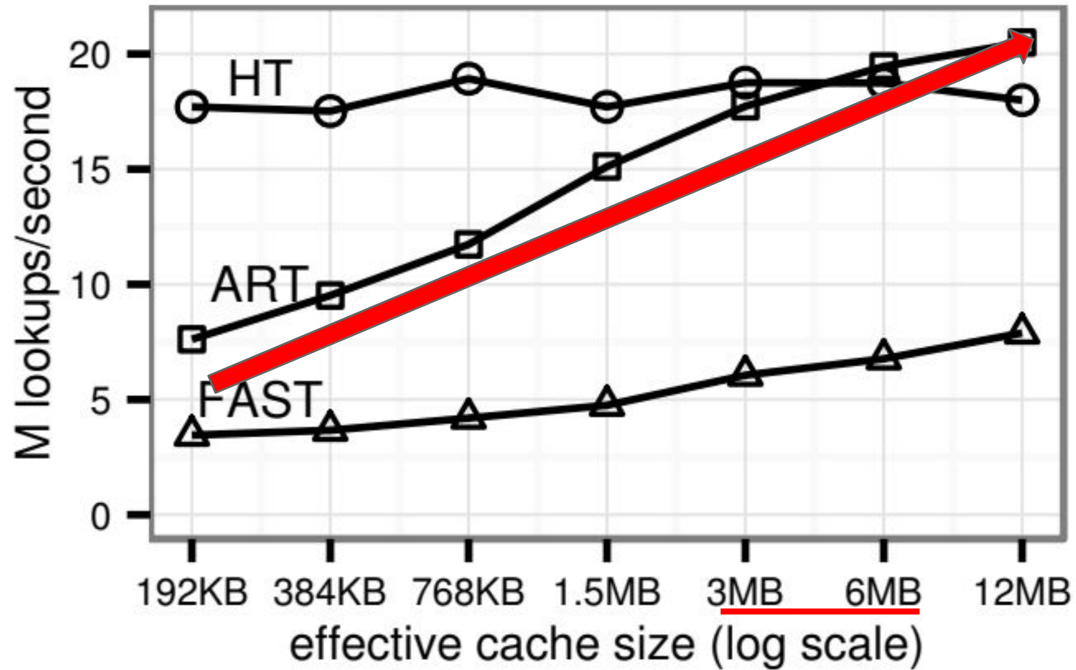
PERFORMANCE COUNTERS PER LOOKUP.

	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

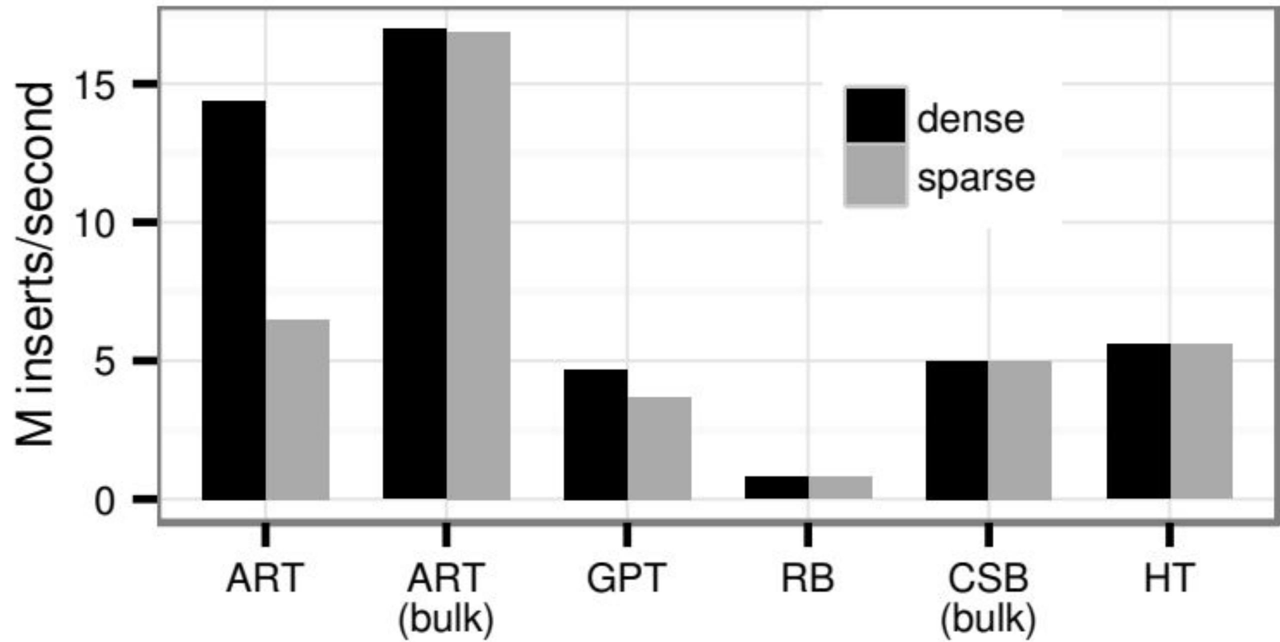
Microbenchmarks - Cache Performance



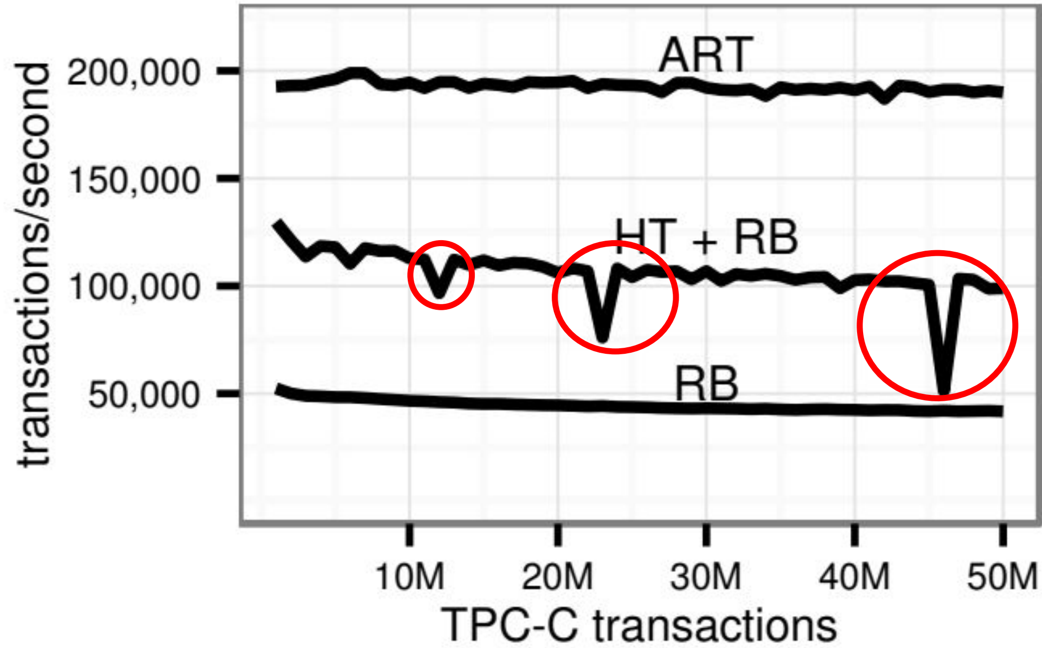
Microbenchmarks - Caching/Skew Effects



Microbenchmarks - Cache Size



Microbenchmarks - Insertions/Updates



Why do we see these drops?

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

TPC-C Space Consumption

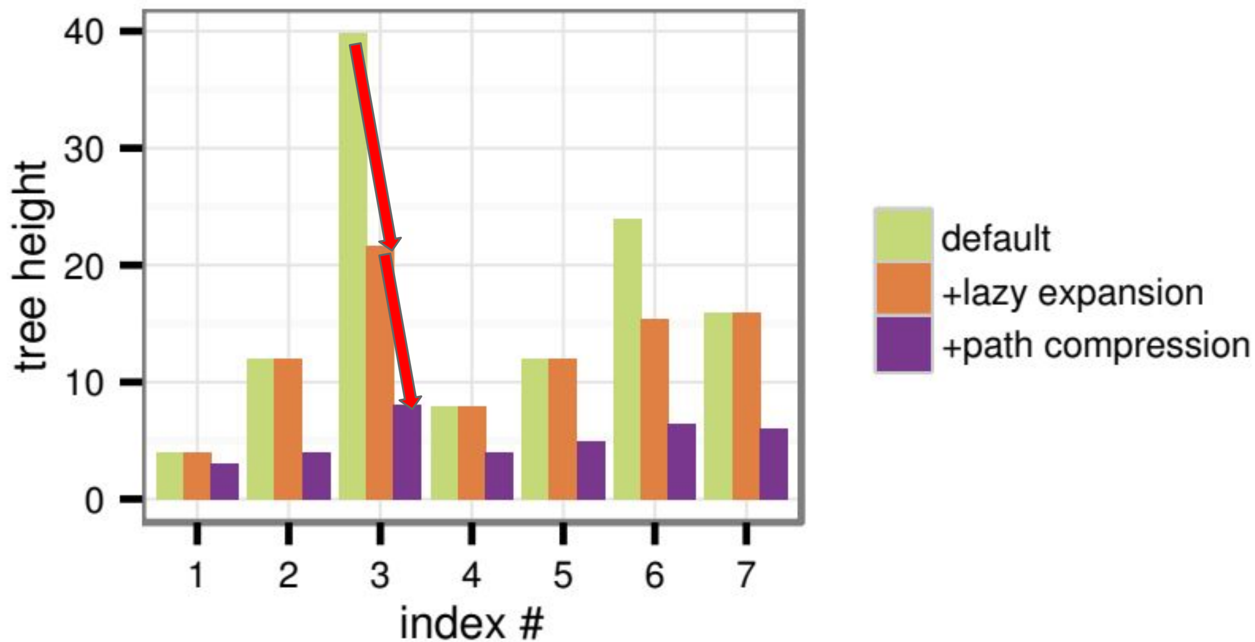


Fig. 17. Impact of lazy expansion and path compression on the height of the TPC-C indexes.

TPC-C Lazy Expansion and Path Compression

Conclusion

- Solved the in-memory DB bottleneck
- Not as relevant today

Pros:

- Thorough problem definition
- Solution addresses every point

Cons:

- Narrow application
- Does not address multithreading
- Is not as effective on disk

Thank You!