

class 11

## Adaptive Radix Trees

Prof. Manos Athanassoulis

<https://bu-disc.github.io/CS561/>

# Indexing is key to database performance

**B+ Trees** dominate disk-based indexes

**Hash tables** and optimized search trees are common for in-memory

***BUT***

**Hash tables** are unordered (no range queries)

**Search trees** are slow

*can we do better?*

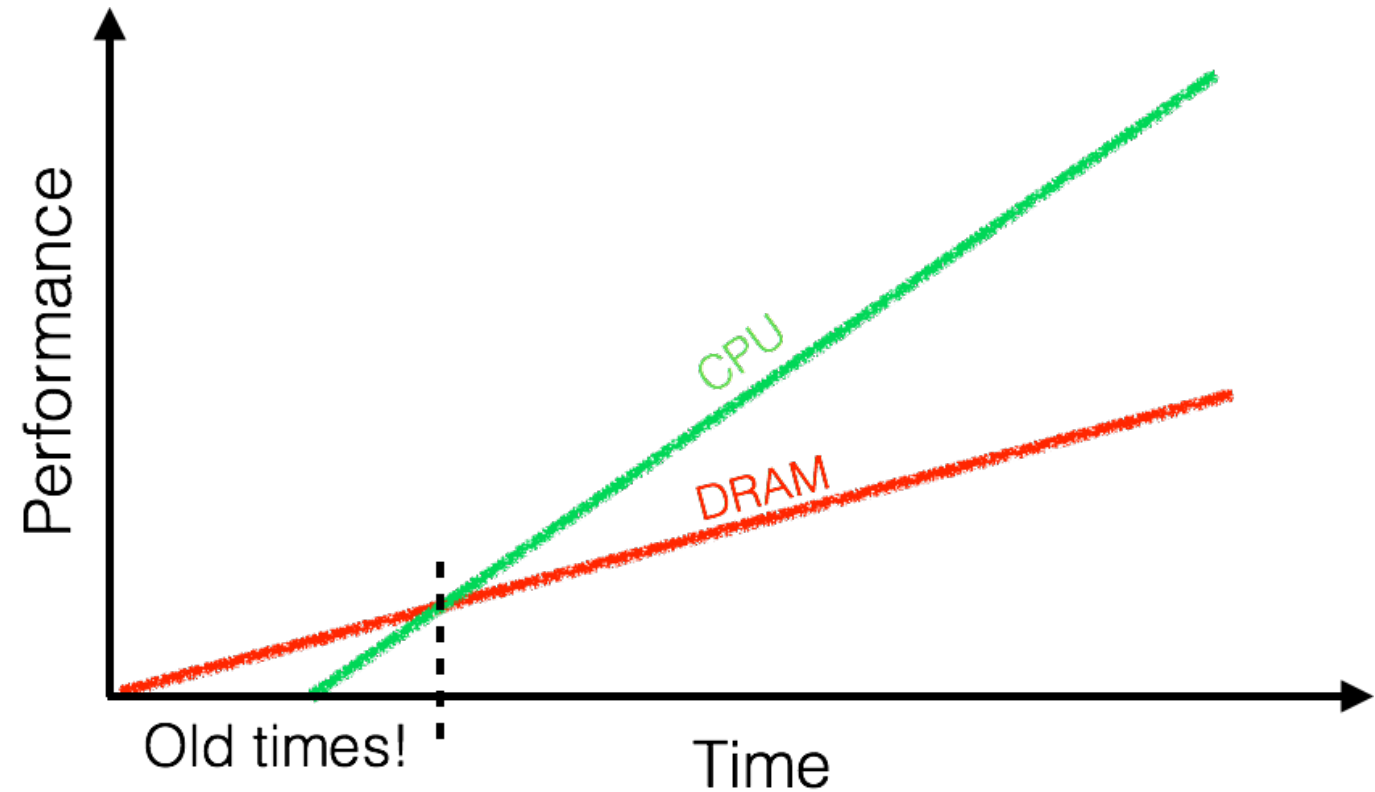
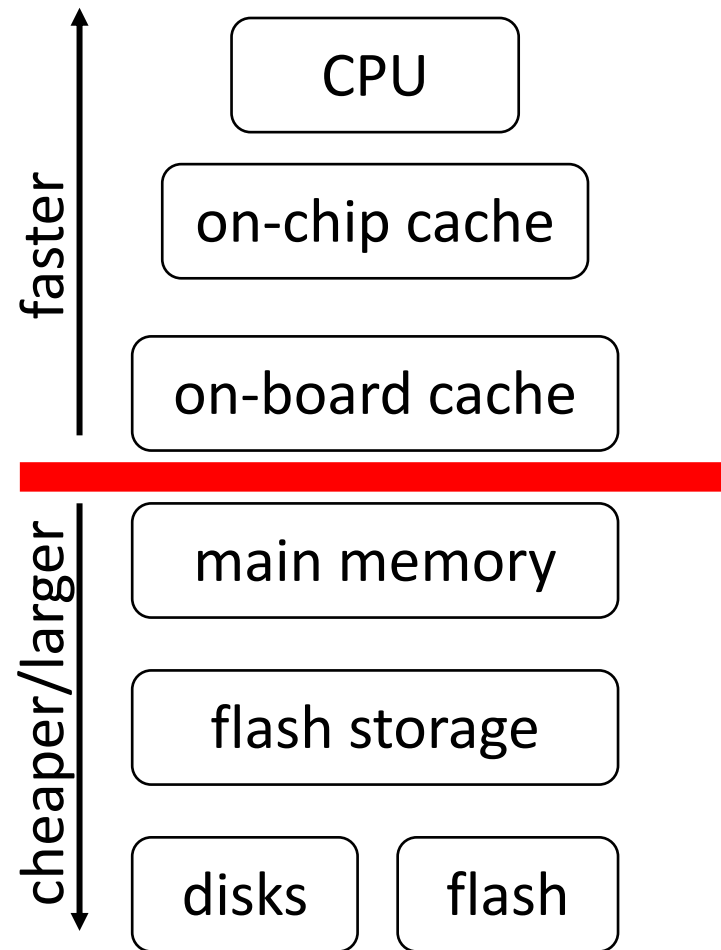
# Increasing data size

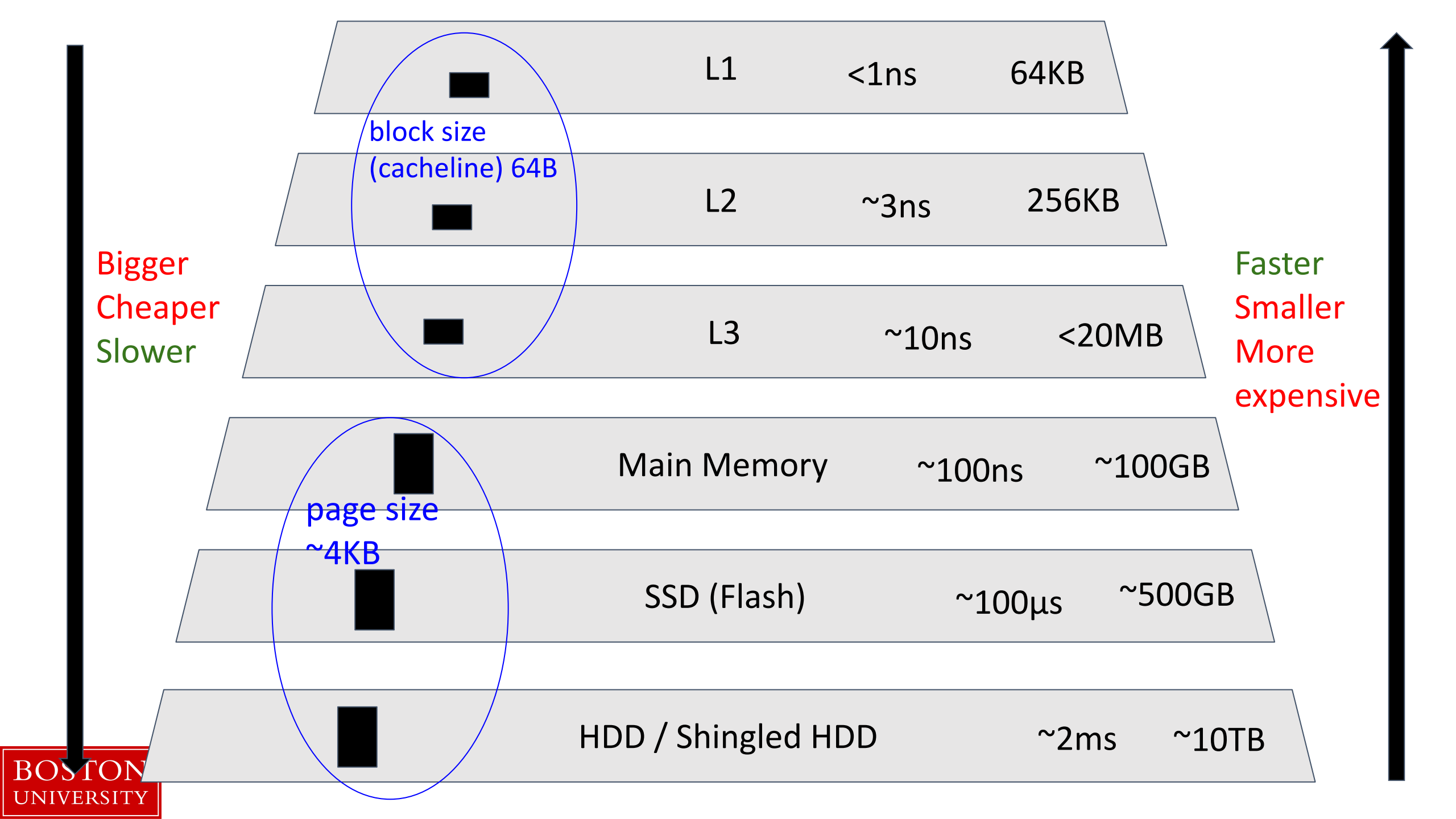
**Search trees size** (tree height and width) grows with data size!

So it quickly does not fit in cache or in memory

*Why is that problem?*

# Reminder: Memory Wall



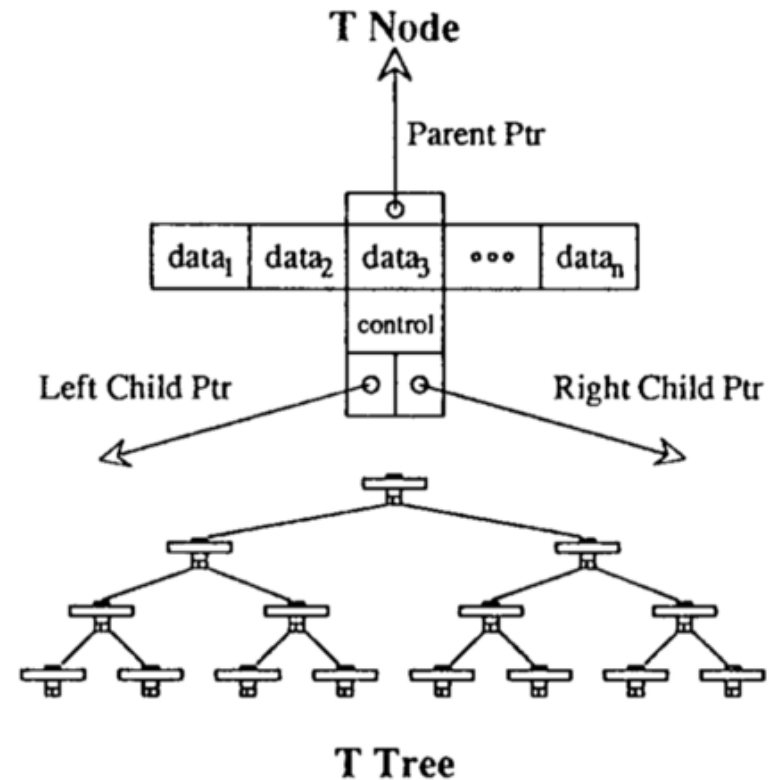


# In-Memory Search Trees: T-Trees

Fat nodes (~cacheline size) with two children

Developed in the 80s (still used in some systems!)

Unpredictable pointer chasing

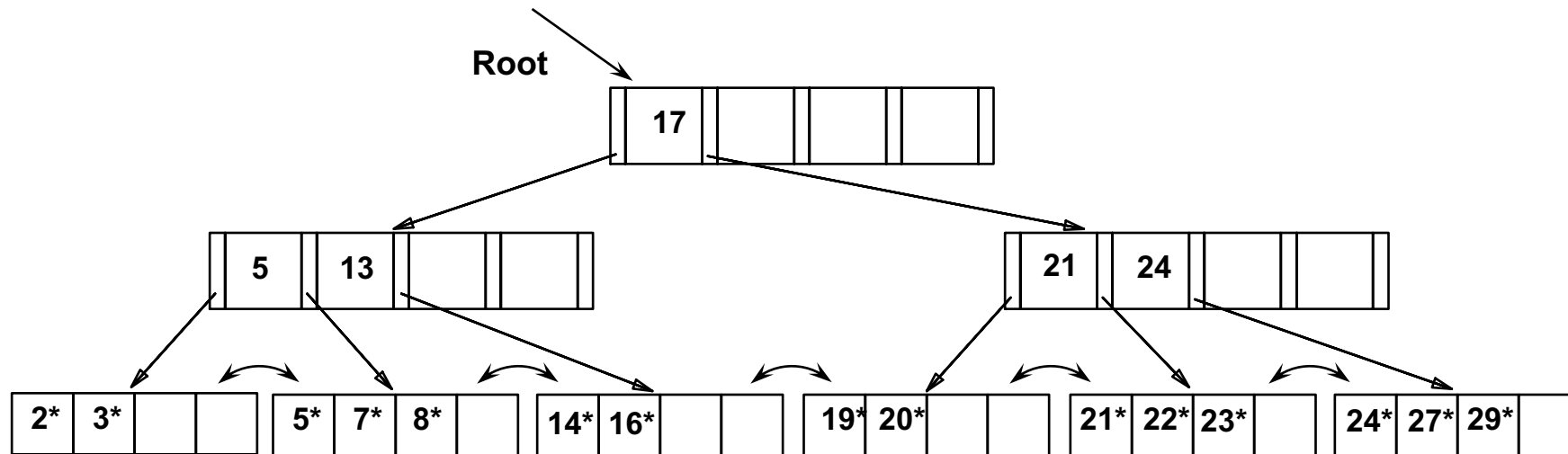


# Are B+ Trees good for in-memory execution?

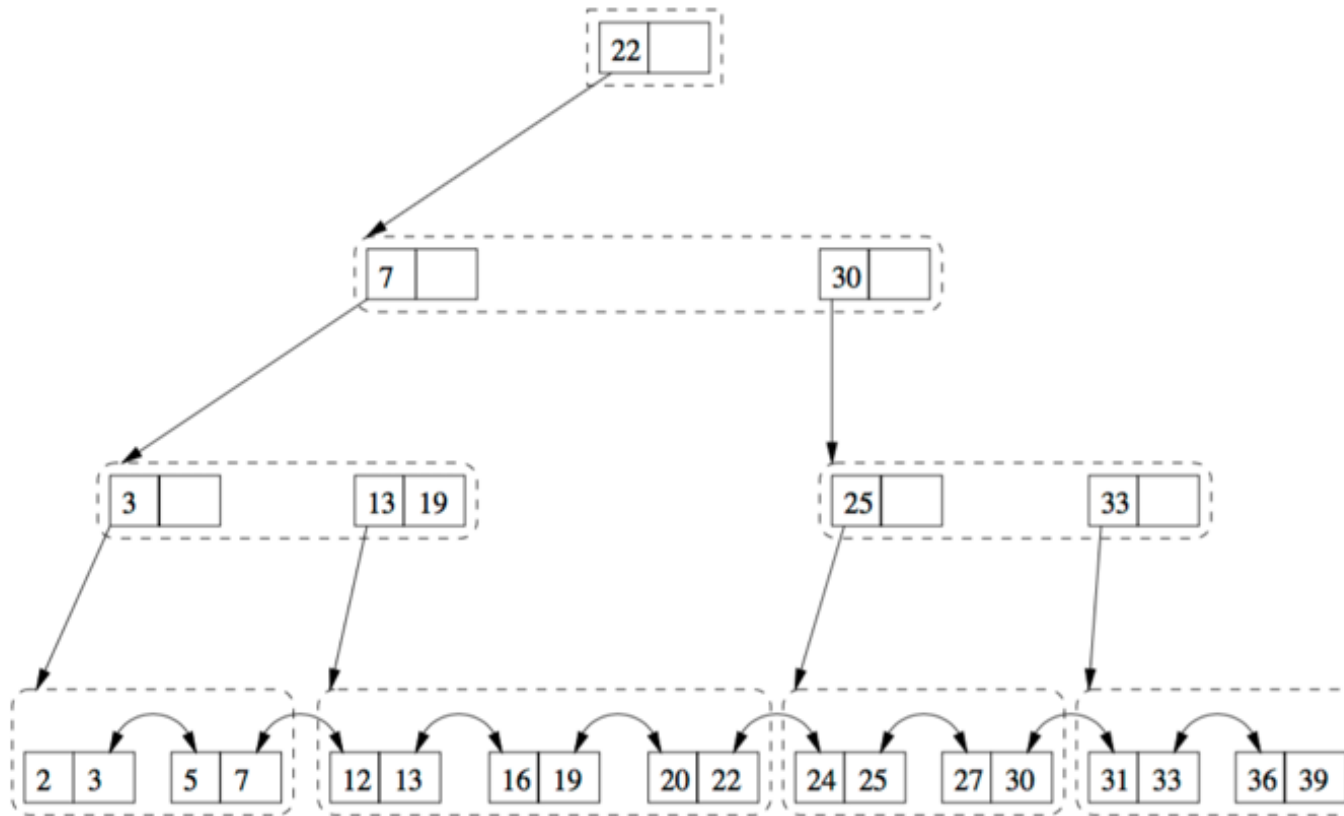
Designed for disks!

Nodes have page size, the goal is to minimize #random accesses of pages (wide fanout)

How to make it memory friendly?



# Cache-sensitive B+ Trees



Every level is physically stored contiguously

Good cache utilization

Poor updates – needs logic to balance

Tree height depends on #items inserted



# Can we do better for an in-memory search tree?

Maintain order

tree

Maintain few random access

low height

Maintain good cache utilization

access cachelines

Maintain low space complexity

Cheap updates

less logic, avoid rebalancing or splitting

# Enter Tries

Also known as Radix Trees, Prefix Trees, Digital Trees

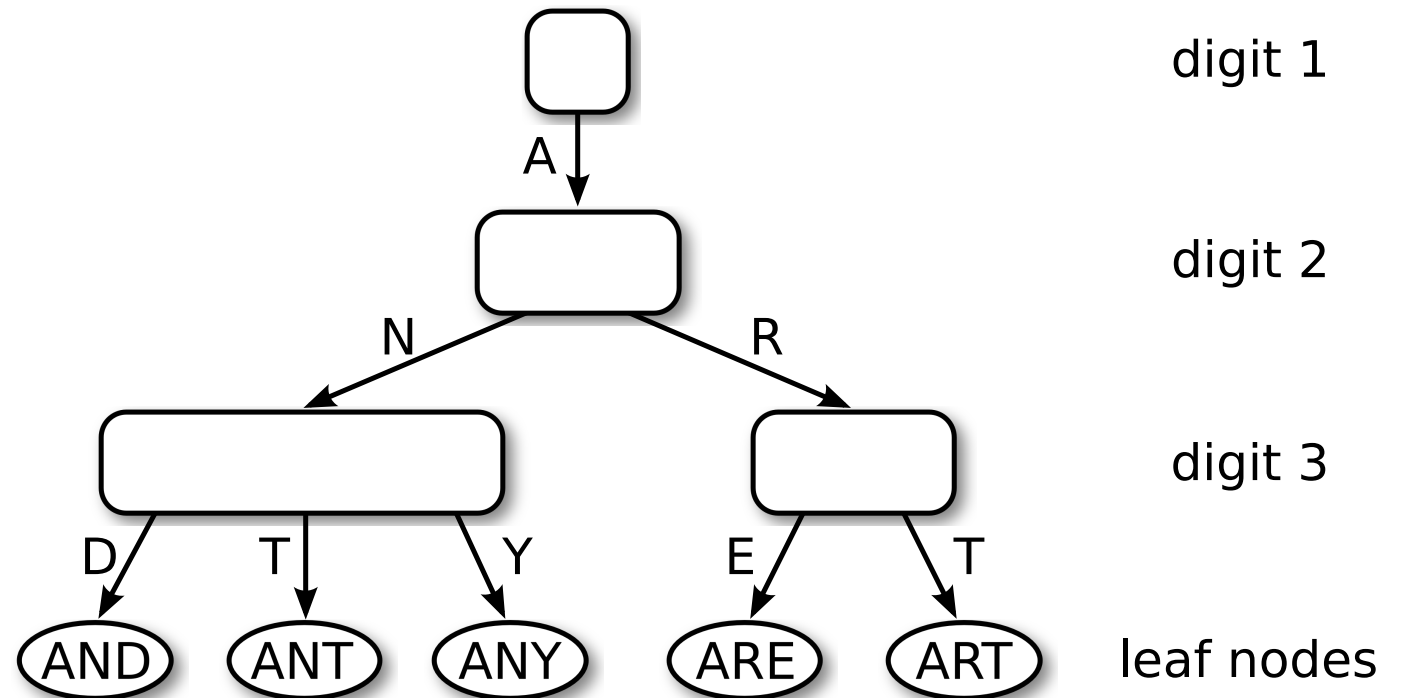
# Trie, Radix Tree, Prefix Tree, Digital Tree

Tree height depends on key length  $k$

Not on tree (data) size

No rebalancing needed!

Automatically get lexicographical order

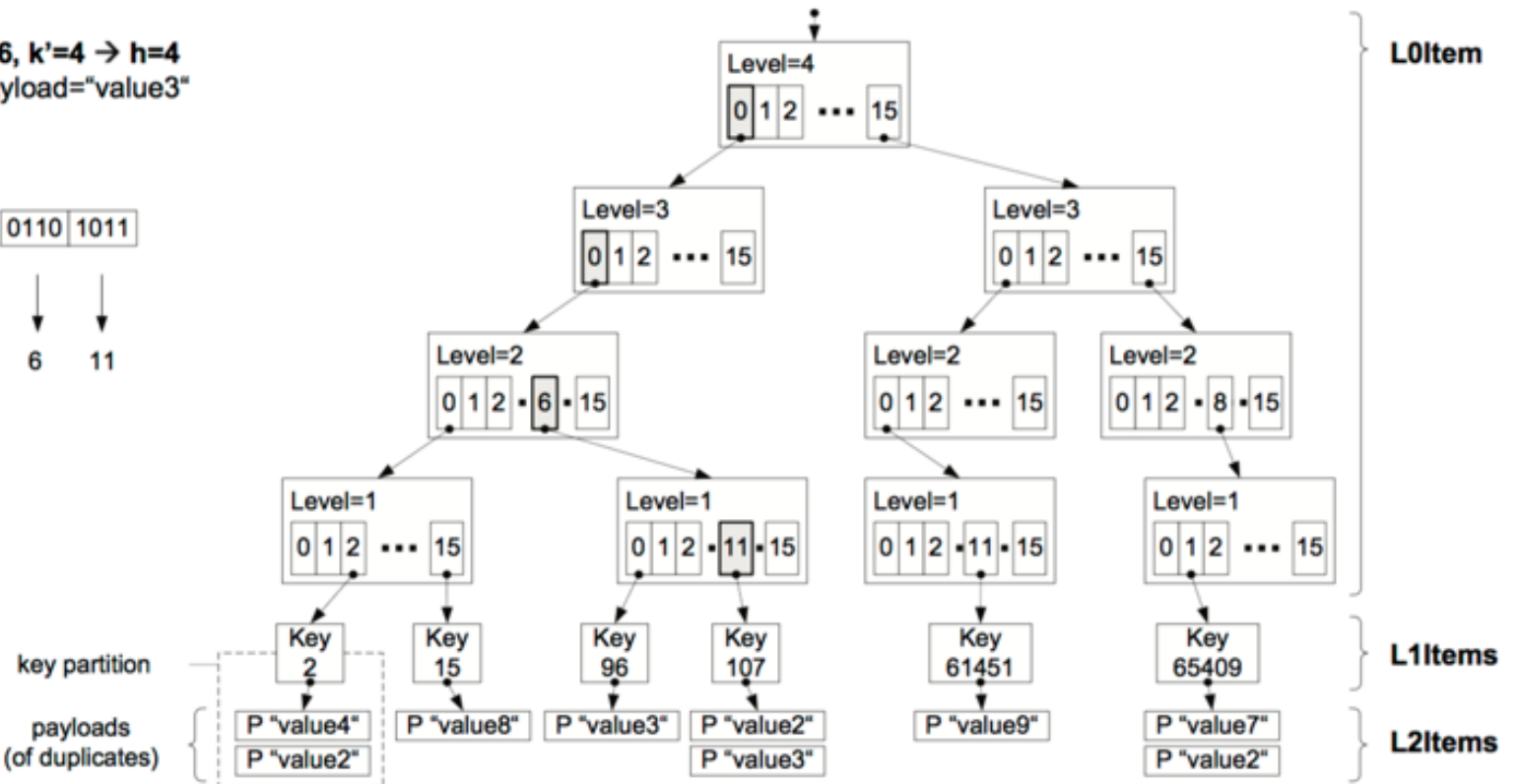


# Tries on integers (in binary format)

Every node stores a part of the binary representation (“radix”) of the key

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

Significant space savings

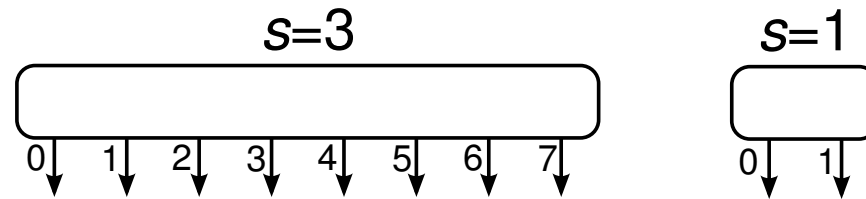
Should all nodes use the same number of radix bits?

# Adaptive Radix Tree Span

For binary keys, the fanout can be configured!

Each node uses  $s$  bits (“span”) of the radix of the key

Hence, an inner is an array of  $2^s$  pointers (with equal number of children)



# Tree Size vs. Span

$k$  bit keys & span= $s \rightarrow k/s$  inner levels &  $2^s$  pointers in each node

32 bit keys & span=1  $\rightarrow$  32 inner levels & 2 pointers in each node

32 bit keys & span=2  $\rightarrow$  16 inner levels & 4 pointers in each node

32 bit keys & span=4  $\rightarrow$  8 inner levels & 16 pointers in each node

32 bit keys & span=8  $\rightarrow$  4 inner levels & 256 pointers in each node

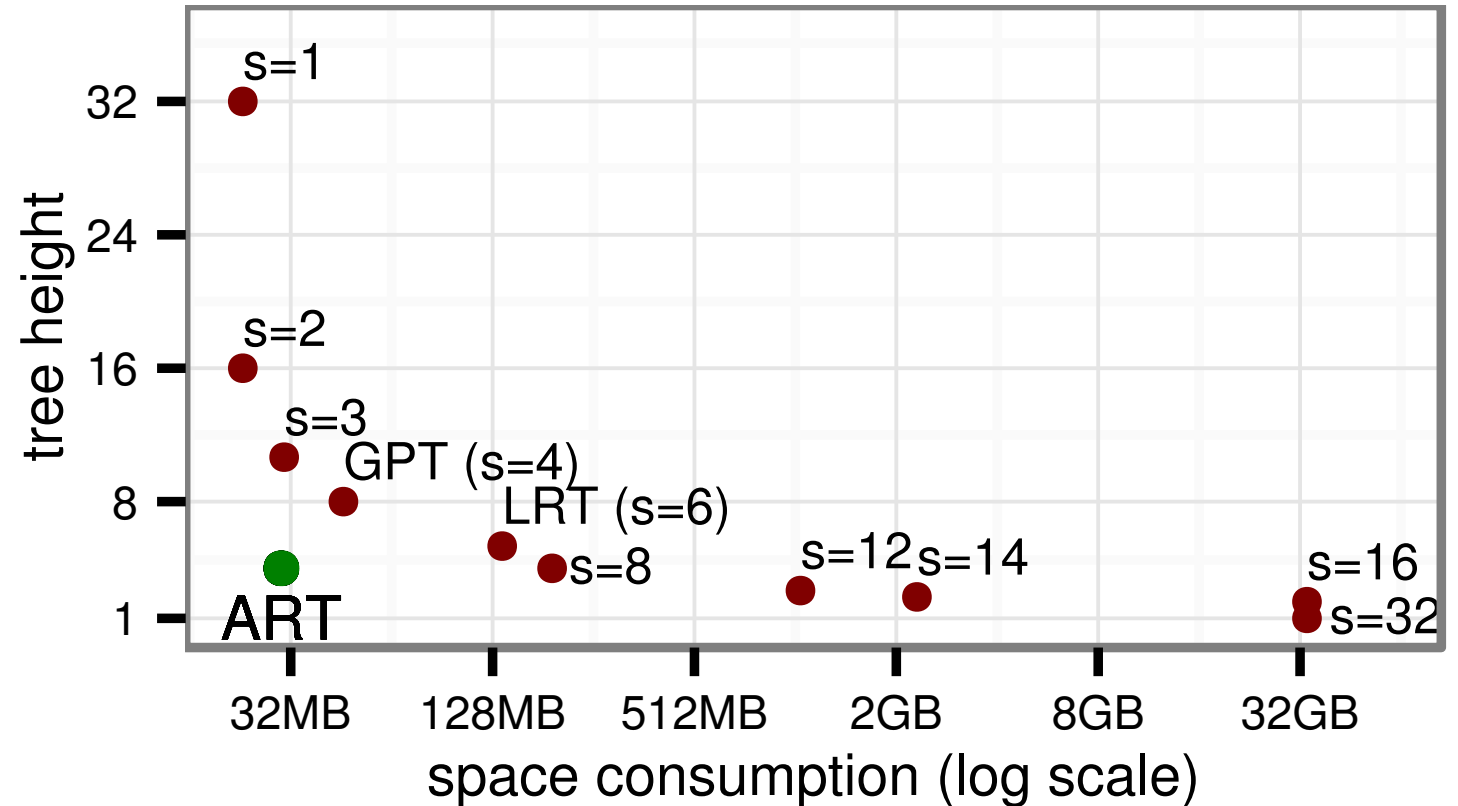
# Height vs. Size Tradeoff

Large  $s$ :  
small height (fast)  
**BUT**  
high space consumption

Small  $s$ :  
large height (slow)  
**BUT**  
low space consumption

ART manages to avoid this tradeoff

How?

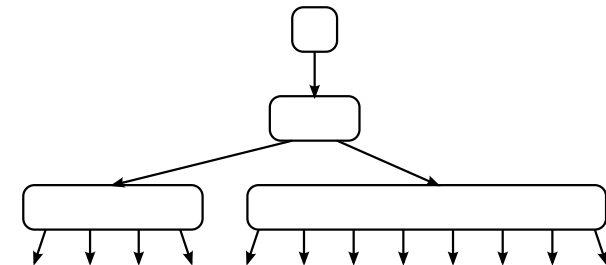
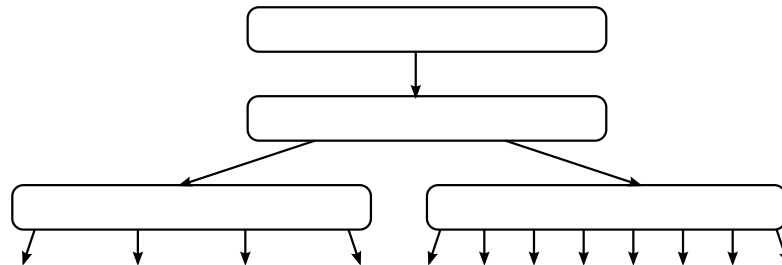


# Adaptively Sized Nodes

$s=8$ : each inner node maps 1 byte of the key to the child node

different node sizes, depending on the number of children

variable fanout





# More on adaptive nodes

4 node sizes, dynamic decision

explicit keys

both Node4 and Node16

use arrays of size 16

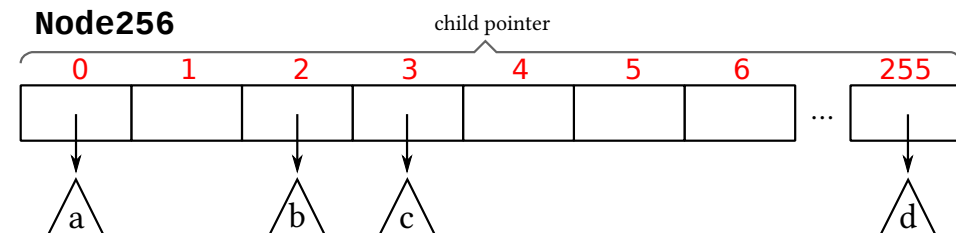
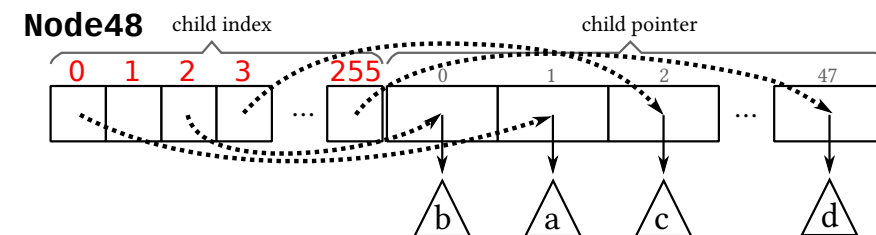
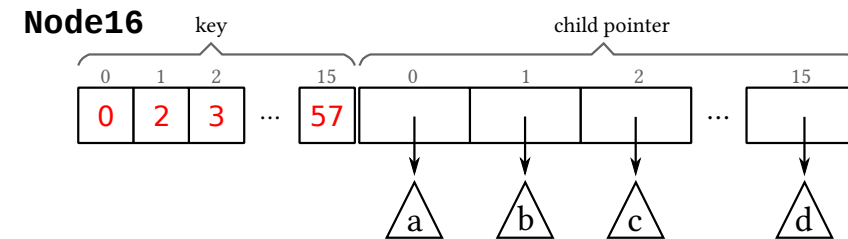
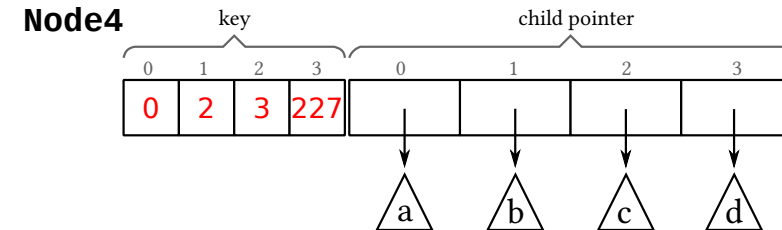
```
typedef struct {  
    art_node n;  
    unsigned char keys[16];  
    art_node *children[16];  
} art_node16;
```

indirection index  
with implicit keys

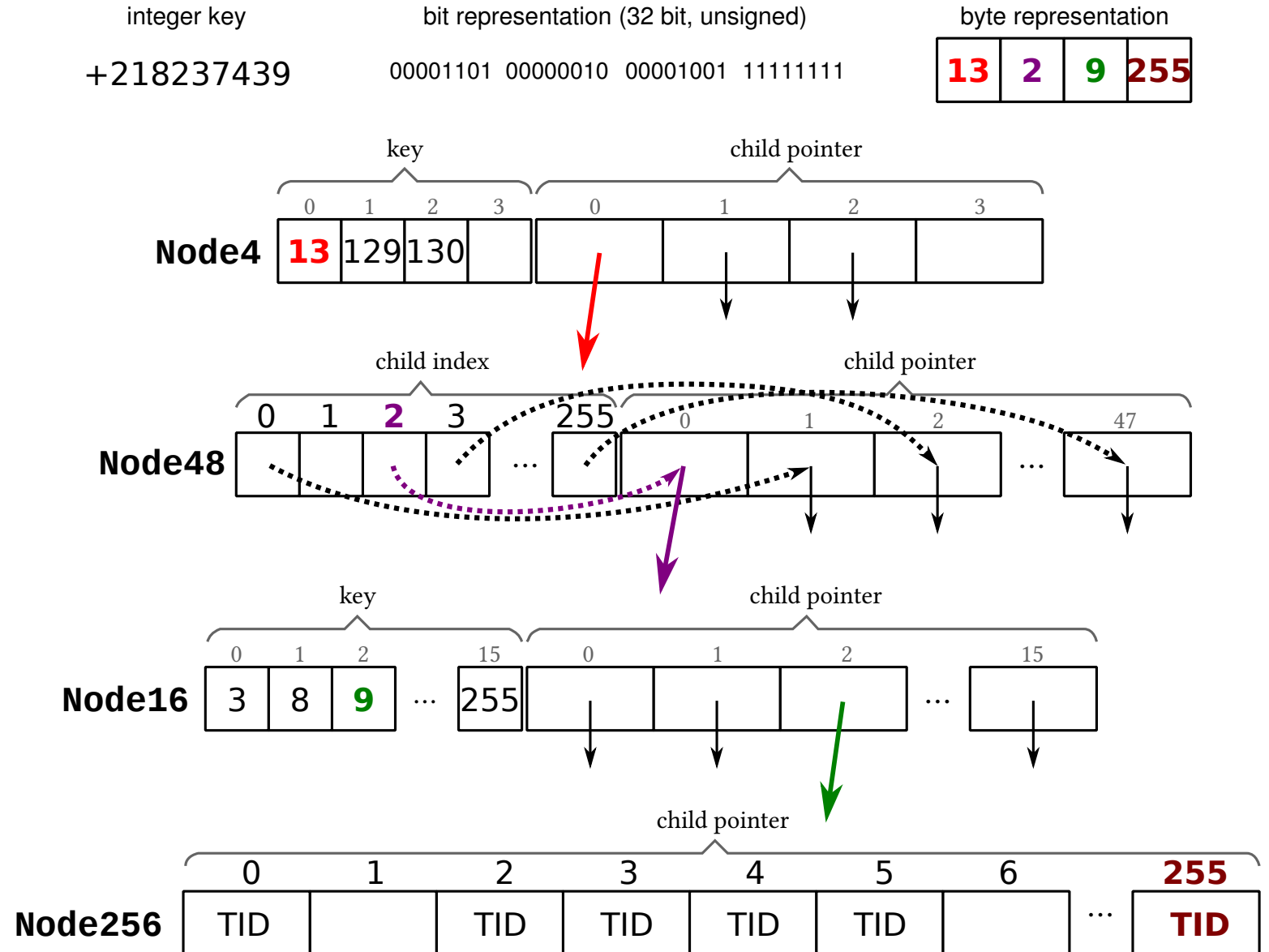
```
typedef struct {  
    art_node n;  
    unsigned char keys[256];  
    art_node *children[48];  
} art_node48;
```

implicit keys

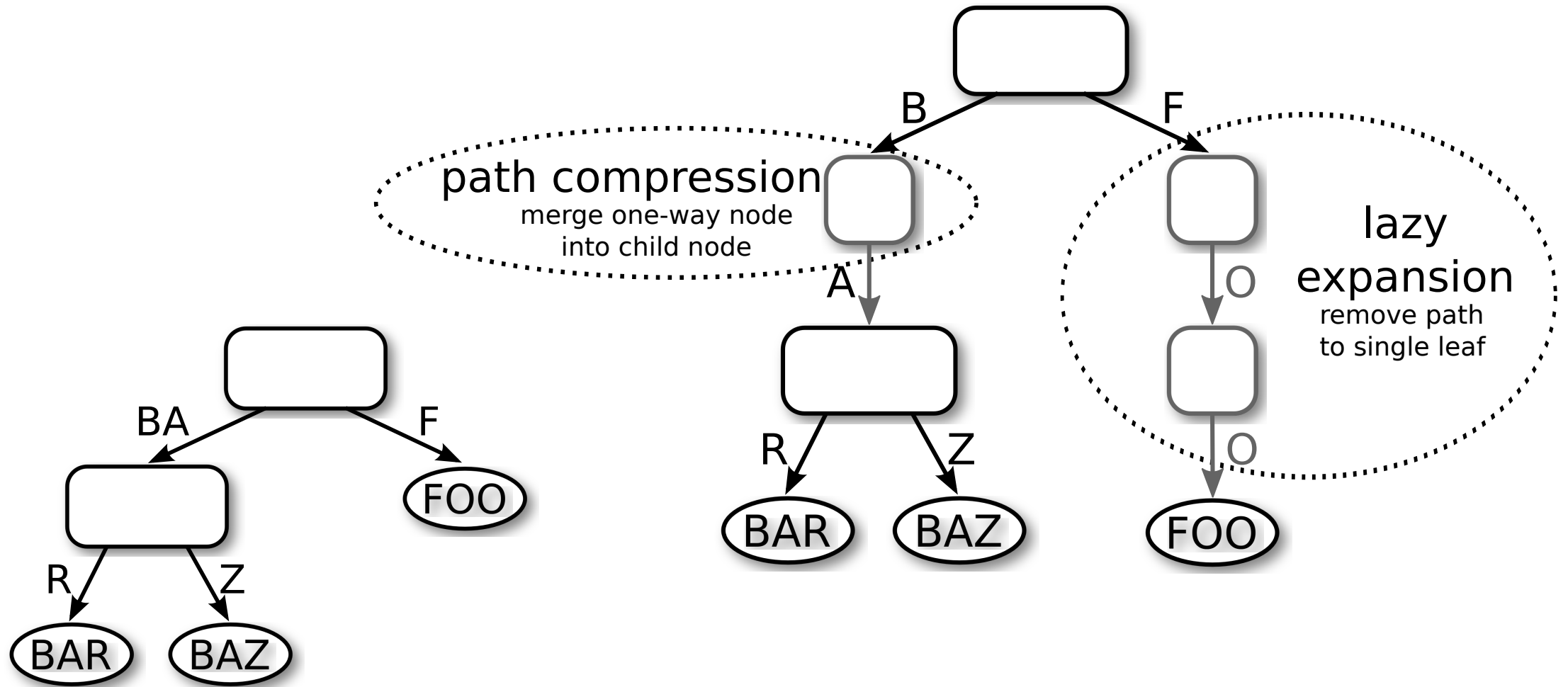
```
typedef struct {  
    art_node n;  
    art_node *children[256];  
} art_node256;
```



# ART Traversal



# Optimizations: Remove one-way nodes



# Supporting various data types

Native support for:

String

Integers (binary representation)

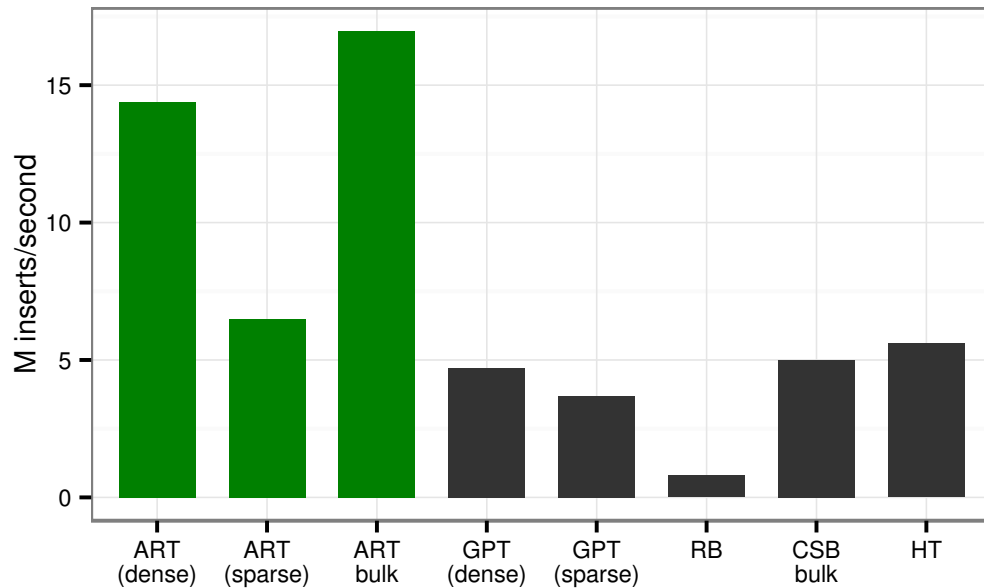
Require transformations for:

floats, Unicode, signed, null, composite

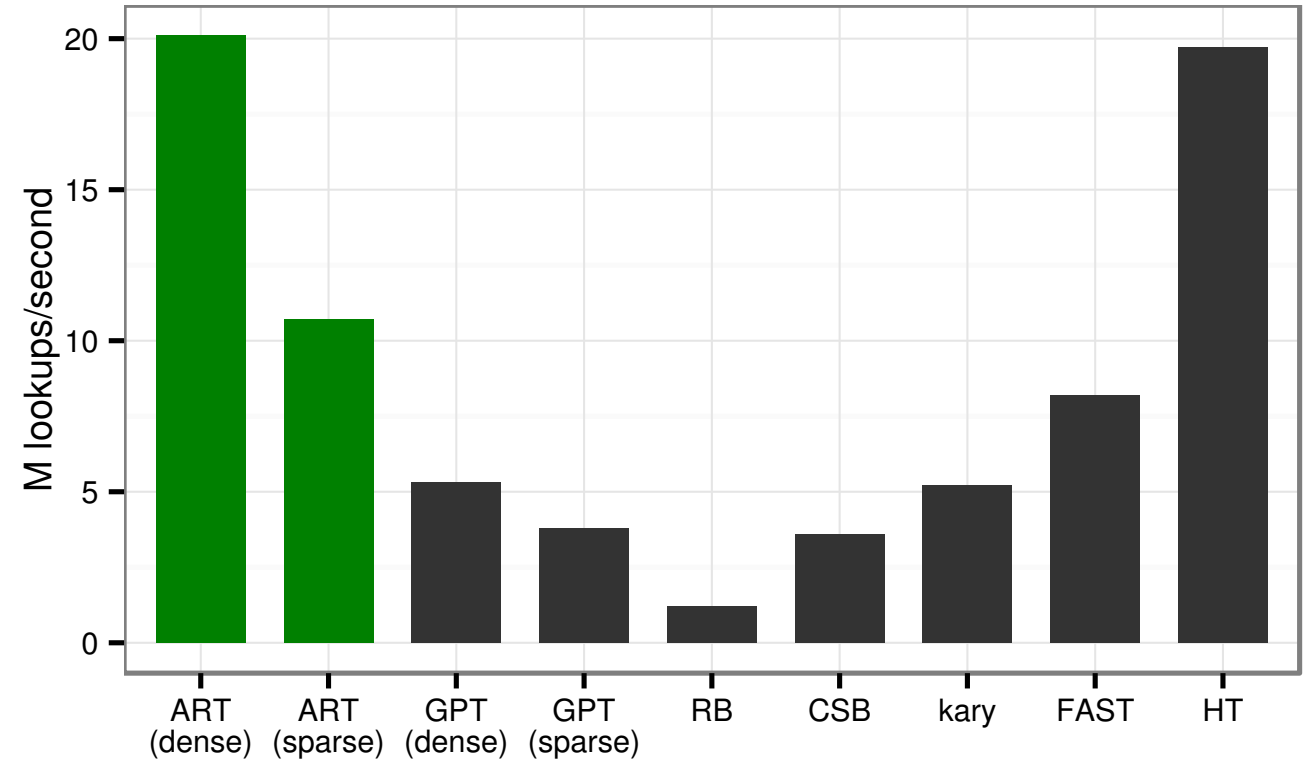
when?

# Evaluation

## Insert performance (4B keys)



## Lookup performance (4B keys)



GPT: Generalized Prefix Tree, Boehm et al., BTW 2011

RB: Red-Black Tree

CSB: Cache-Sensitive B+Tree, Rao and Ross, SIGMOD 2000

kary: K-ary Search Tree, Schlegel et al., Damon 2009

FAST: Fast Architecture Sensitive Tree, Kim et al., SIGMOD 2010

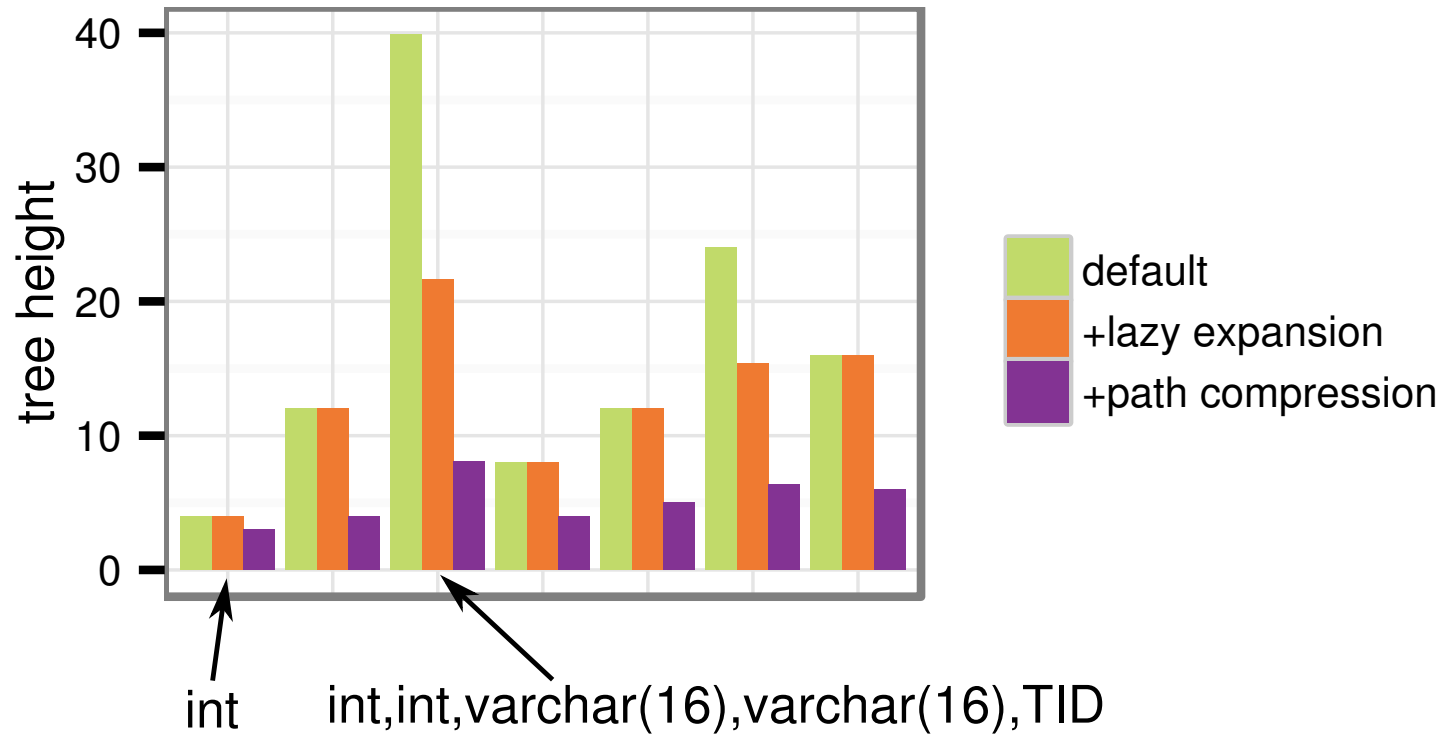
HT: Chained Hash Table

# Cache Efficiency

PERFORMANCE COUNTERS PER LOOKUP.

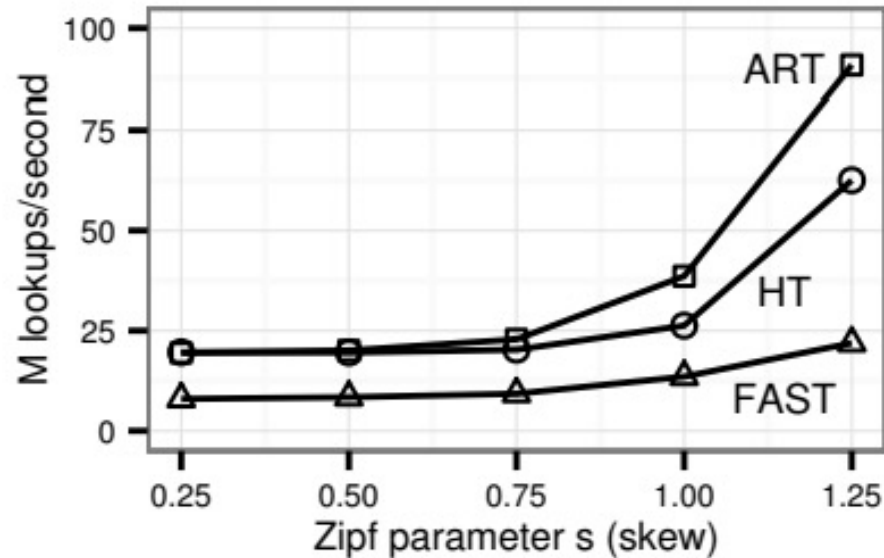
	65K			16M		
	ART (d./s.)	FAST	HT	ART (d./s.)	FAST	HT
Cycles	40/105	94	44	188/352	461	191
Instructions	85/127	75	26	88/99	110	26
Misp. Branches	0.0/0.85	0.0	0.26	0.0/0.84	0.0	0.25
L3 Hits	0.65/1.9	4.7	2.2	2.6/3.0	2.5	2.1
L3 Misses	0.0/0.0	0.0	0.0	1.2/2.6	2.4	2.4

# Reducing Tree Height in TPCC



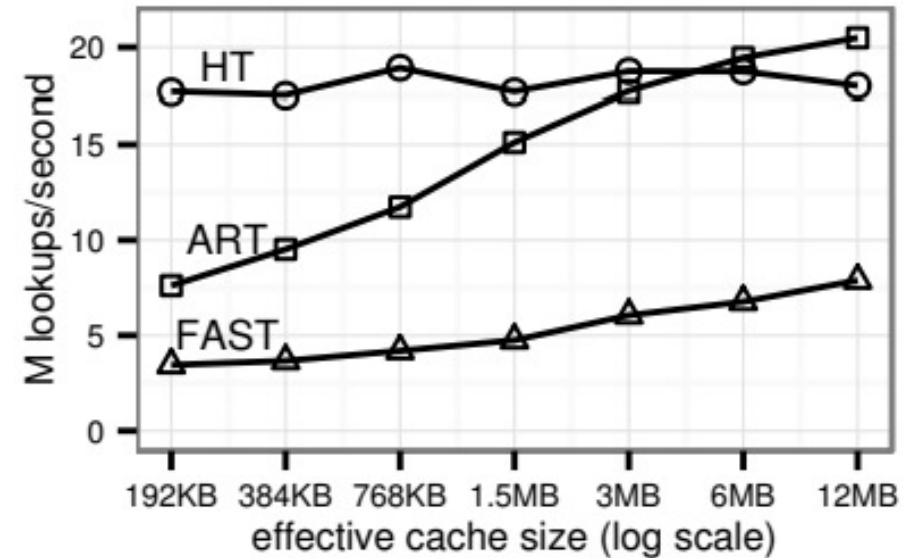
Without the height optimization the height can be the length of the keys → can be prohibitively high

# Skewed Search & Impact of Cache Size



ART: adjacent items are in the same node/subtree

HT: adjacent items are in different buckets



ART: no evictions, fewer misses overall

HT: data is randomly distributed more misses



# Space Efficiency for TPCC

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

# Conclusions

Radix Trees can be used as a generalized index

- for multiple data types

- space efficient

- with excellent performance

thus, combining the benefit of search trees (supports ranges) with lookup efficiency of hash tables

class 11

## Adaptive Radix Trees

Prof. Manos Athanassoulis

<https://bu-disc.github.io/CS561/>