

CS 7270: Advanced Database Systems Fall 2025

Lecture 05

In-memory indexing (Tries)

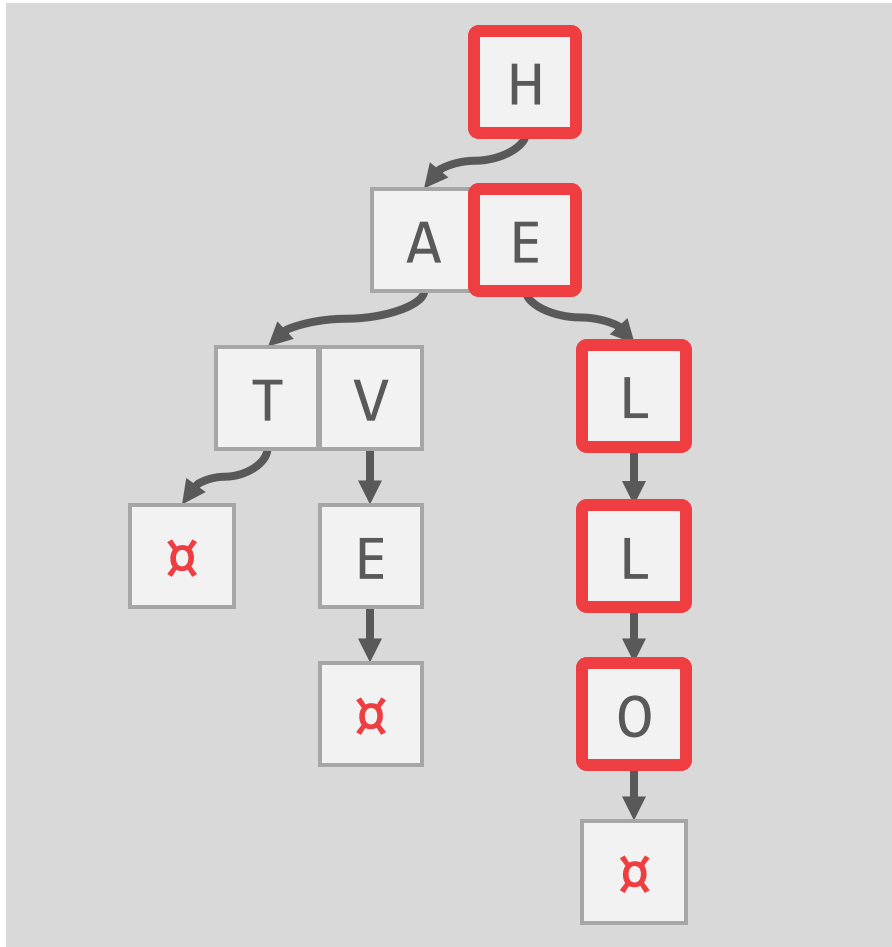
Prashant Pandey

p.pandey@northeastern.edu

Acknowledgement: Slides taken from Prof. Andy Pavlo, CMU

Trie index

Keys: HELLO, HAT, HAVE



- Use a digital representation of keys to examine prefixes one-by-one instead of comparing entire key.
 - Also known as ***Digital Search Tree***, ***Prefix Tree***.

Trie index properties

- Shape only depends on key space and lengths.
 - Does not depend on existing keys or insertion order.
 - Does not require rebalancing operations.
- All operations have $O(k)$ complexity where k is the length of the key.
 - The path to a leaf node represents the key of the leaf
 - Keys are stored implicitly and can be reconstructed from paths.

Trie index properties

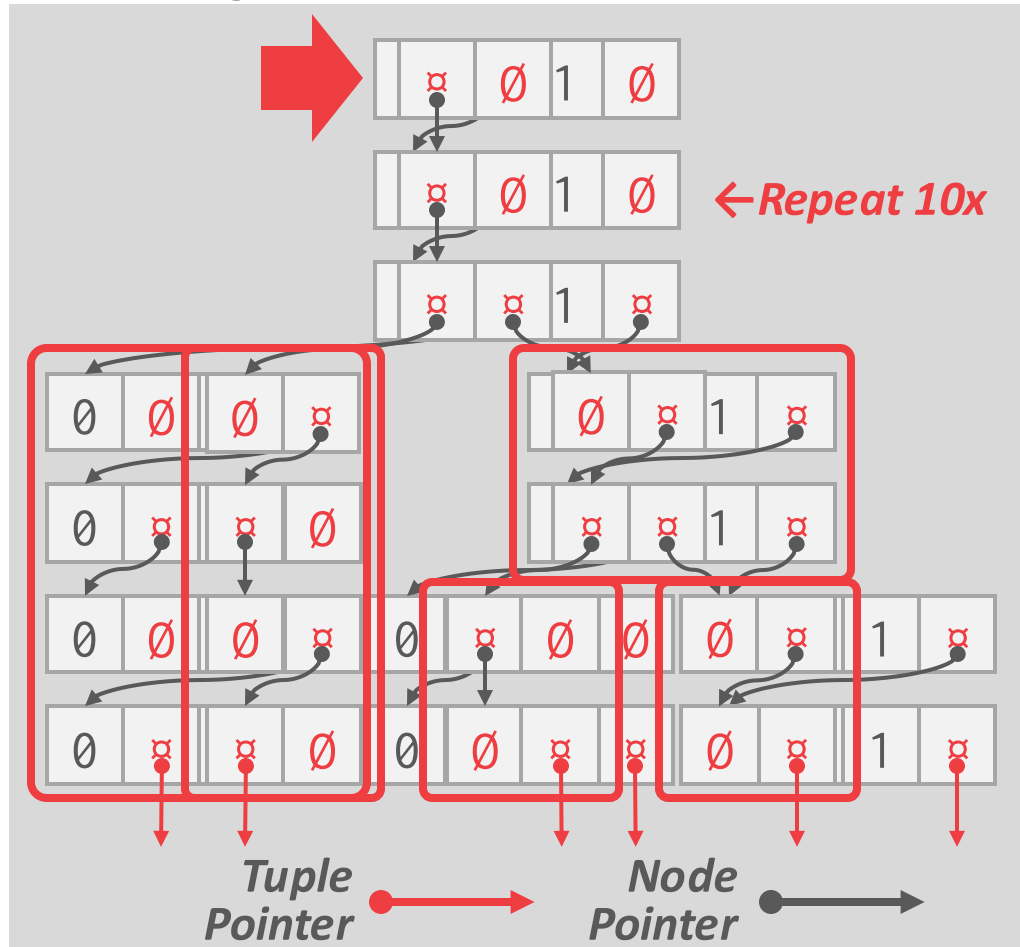
- Shape only depends on key space and lengths.
 - Does not depend on existing keys or insertion order.
 - Does not require rebalancing operations.
 - All operations have $O(k)$ complexity where k is the length of the key.
 - The path to a leaf node represents the key of the leaf
 - Keys are stored implicitly and can be reconstructed from paths.
- History independent** ←

Trie key span

- The **span** of a trie level is the number of bits that each partial key / digit represents.
 - If the digit exists in the corpus, then store a pointer to the next level in the trie branch. Otherwise, store null.
- This determines the **fan-out** of each node and the physical **height** of the tree.
 - *n*-way Trie = Fan-Out of *n*

Trie key span

1-bit Span Trie



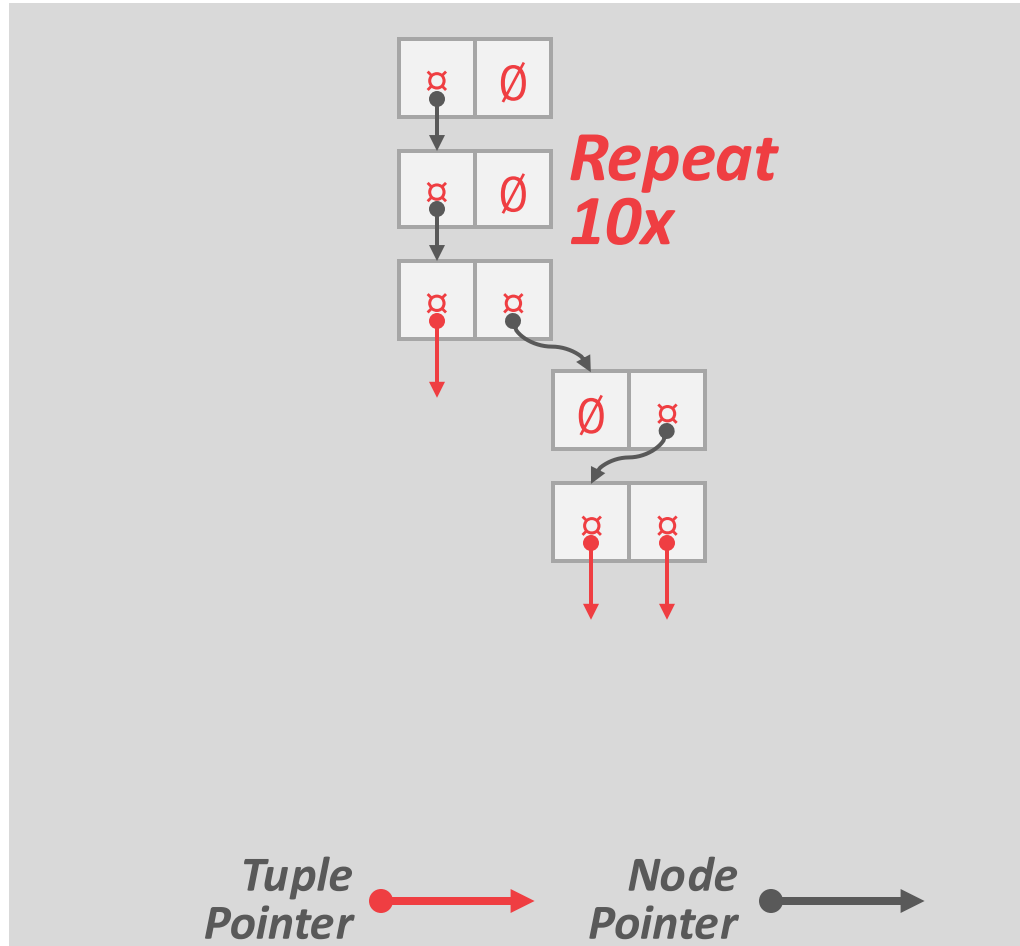
A red arrow points down to the following key spans:

K10 → 00000000 00001010
K25 → 00000000 00011001
K31 → 00000000 00011111

Red boxes highlight the first bit of the first byte (0) and the last three bits of the second byte (1010, 1001, 1111) for each key.

Radix tree

1-bit Span Radix Tree



- Omit all nodes with only a single child.
 - Also known as **Patricia Tree**.
- Can produce false positives, so the DBMS always checks the original tuple to see whether a key matches.

Trie variants

- Judy Arrays (HP)
- ART Index (HyPer)
- Masstree (Silo)

Judy arrays

- Variant of a 256-way radix tree. First known radix tree that supports adaptive node representation.
- Three array types
 - **Judy1**: Bit array that maps integer keys to true/false.
 - **JudyL**: Map integer keys to integer values.
 - **JudySL**: Map variable-length keys to integer values.
- Open-Source Implementation (LGPL).
Patented by HP in 2000. Expires in 2022.
 - Not an issue according to authors.
 - <http://judy.sourceforge.net/>

Judy arrays

- Do not store meta-data about node in its header.
 - This could lead to additional cache misses.
- Pack meta-data about a node in 128-bit "Judy Pointers" stored in its parent node.
 - Node Type
 - Population Count
 - Child Key Prefix / Value (if only one child below)
 - 64-bit Child Pointer



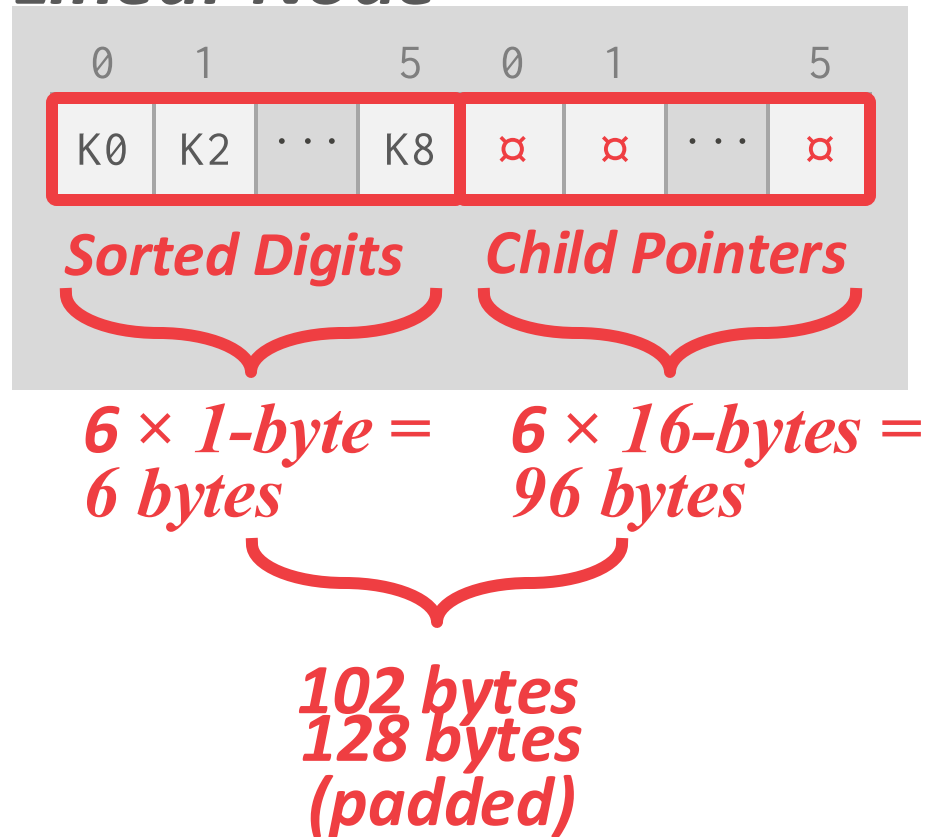
Judy arrays: node types

- Every node can store up to 256 digits.
- Not all nodes will be 100% full though.
- Adapt node's organization based on its keys.
 - **Linear Node:** Sparse Populations
 - **Bitmap Node:** Typical Populations
 - **Uncompressed Node:** Dense Population



Judy arrays: Linear nodes

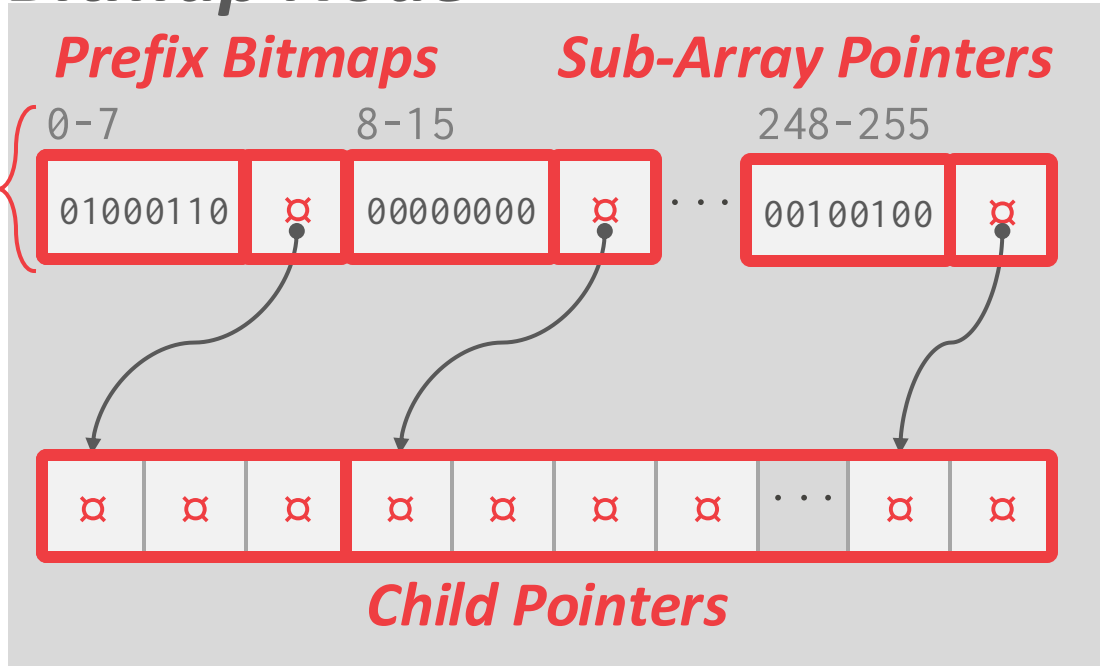
Linear Node



- Store sorted list of partial prefixes up to two cache lines.
 - Original spec was one cache line
- Store separate array of pointers to children ordered according to prefix sorted.

Judy arrays: bitmap nodes

Bitmap Node



- 256-bit map to mark whether a prefix is present in node.
- Bitmap is divided into eight segments, each with a pointer to a sub-array with pointers to child nodes.

Digit	
0	→00000000
1	→00000001
2	→00000010
3	→00000011
4	→00000100
5	→00000101
6	→00000110
7	→00000111

Adaptive radix tree (ART)

- Developed for TUM HyPer DBMS in 2013.
- 256-way radix tree that supports different node types based on its population.
 - Stores meta-data about each node in its header.
- Concurrency support was added in 2015.

ART vs. JUDY

- **Difference #1: Node Types**

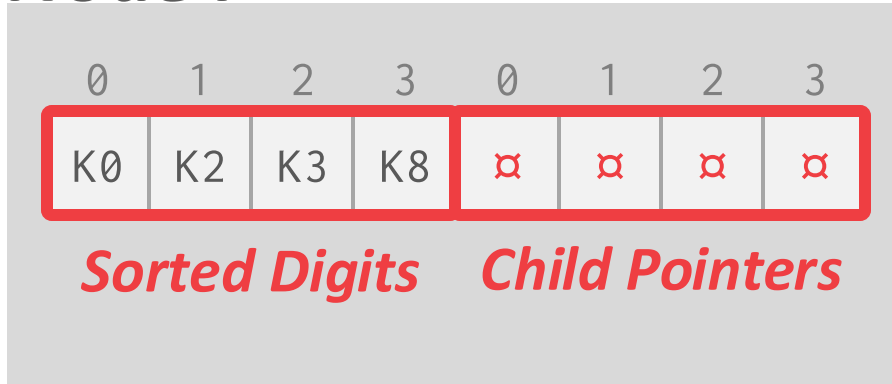
- Judy has three node types with different organizations.
- ART has four nodes types that (mostly) vary in the maximum number of children.

- **Difference #2: Purpose**

- Judy is a general-purpose associative array. It "owns" the keys and values.
- ART is a table index and does not need to cover the full keys. Values are pointers to tuples.

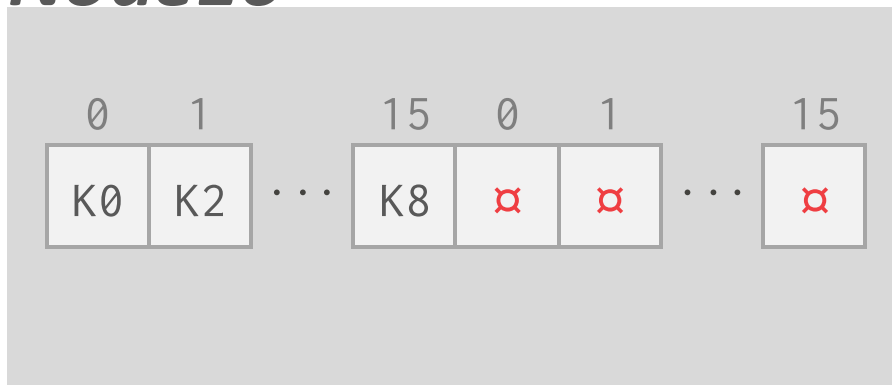
ART: inner node types (1)

Node4



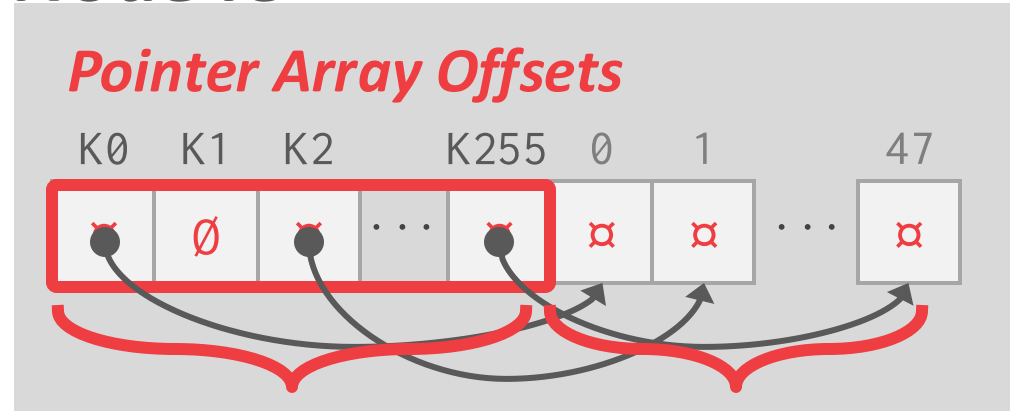
- Store only the 8-bit digits that exist at a given node in a sorted array.
- The offset in sorted digit array corresponds to offset in value array.

Node16



ART: inner node types (2)

Node48



$256 \times 1\text{-byte}$
 $= 256 \text{ bytes}$

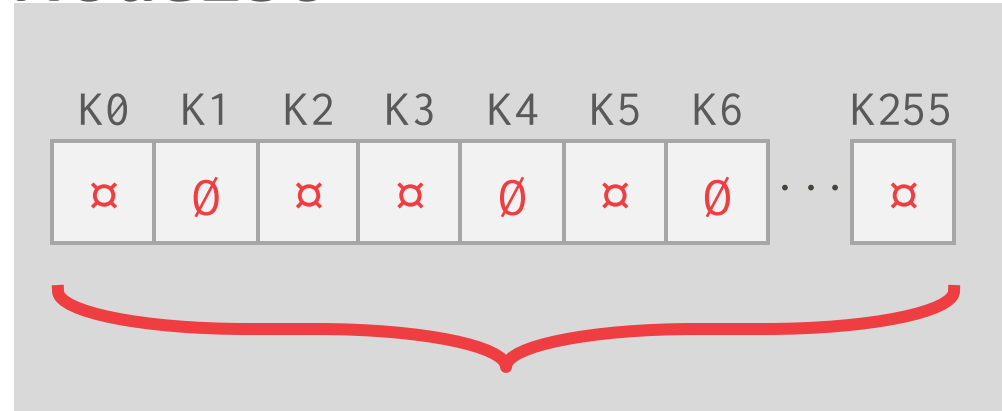
$48 \times 8\text{-bytes} =$
 384 bytes

640 bytes

- Instead of storing 1-byte digits, maintain an array of 1-byte offsets to a child pointer array that is indexed on the digit bits.

ART: inner node types (3)

Node256



*256 × 8-byte
= 2048 bytes*

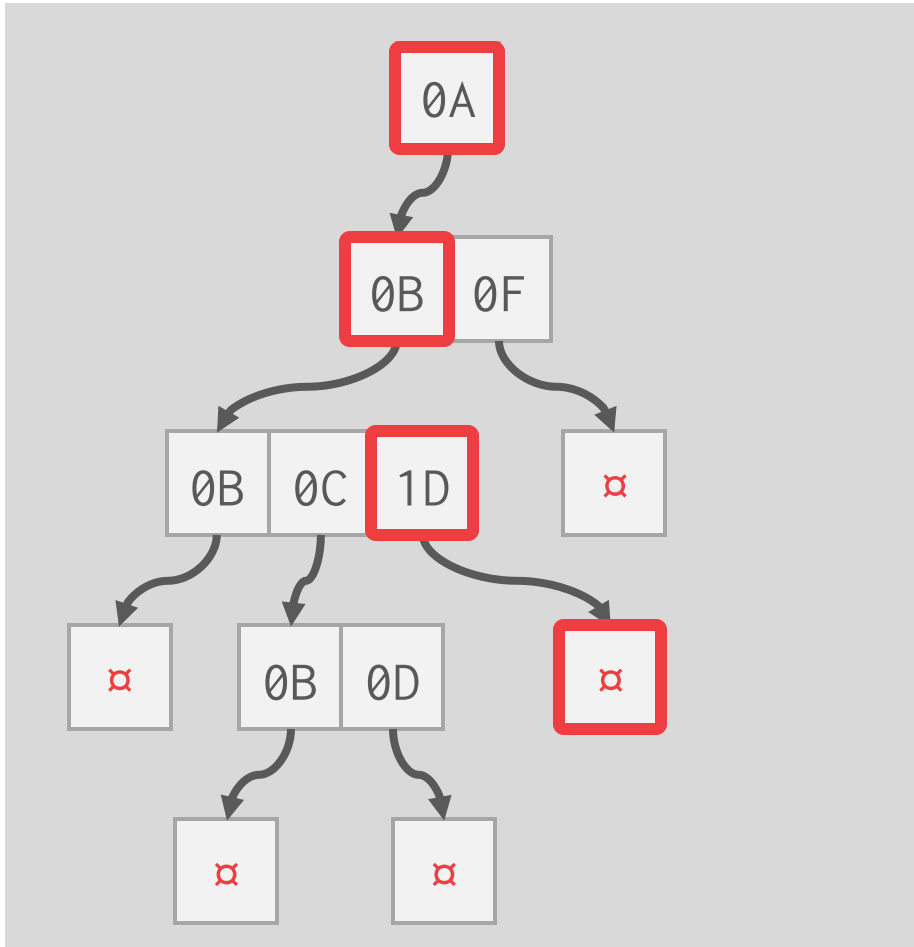
- Store an array of 256 pointers to child nodes. This covers all possible values in 8-bit digits.
- Same as the Judy Array's Uncompressed Node.

ART: binary comparable keys

- Not all attribute types can be decomposed into binary comparable digits for a radix tree.
 - **Unsigned Integers:** Byte order must be flipped for little endian machines.
 - **Signed Integers:** Flip two's-complement so that negative numbers are smaller than positive.
 - **Floats:** Classify into group (neg vs. pos, normalized vs. denormalized), then store as unsigned integer.
 - **Compound:** Transform each attribute separately.

ART: binary comparable keys

8-bit Span Radix Tree



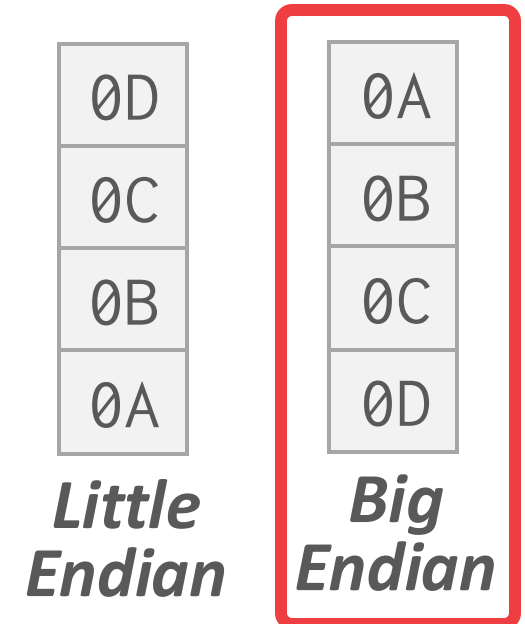
Int Key: 168496141



Hex Key: 0A 0B 0C 0D

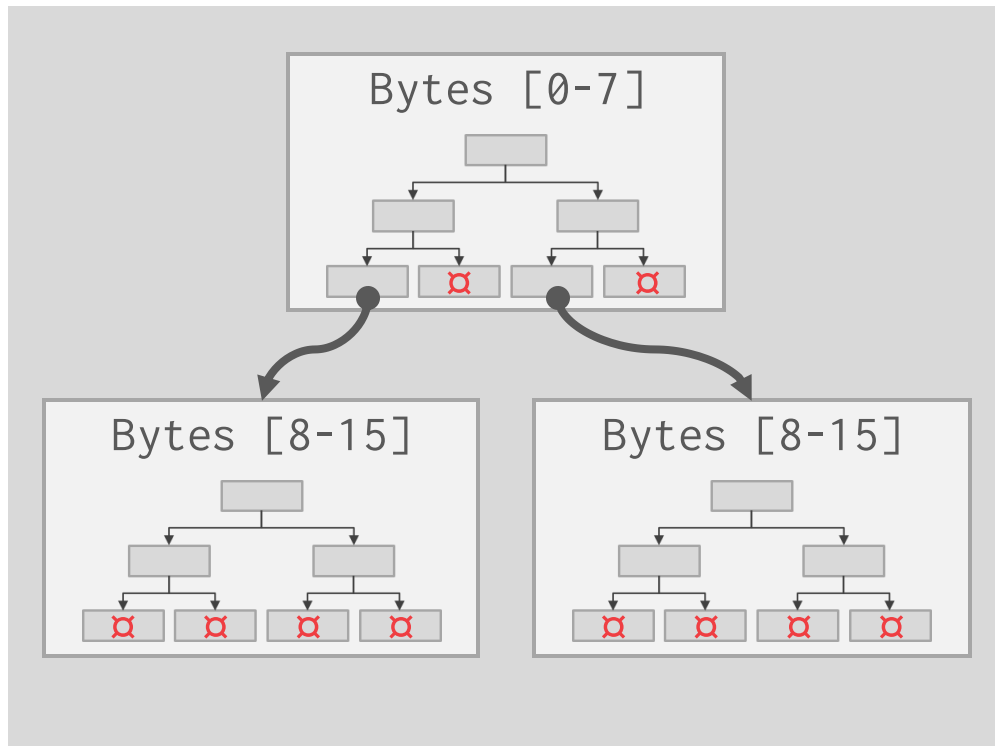
Find: 658205

Hex: 0A 0B 1D



MASSTREE

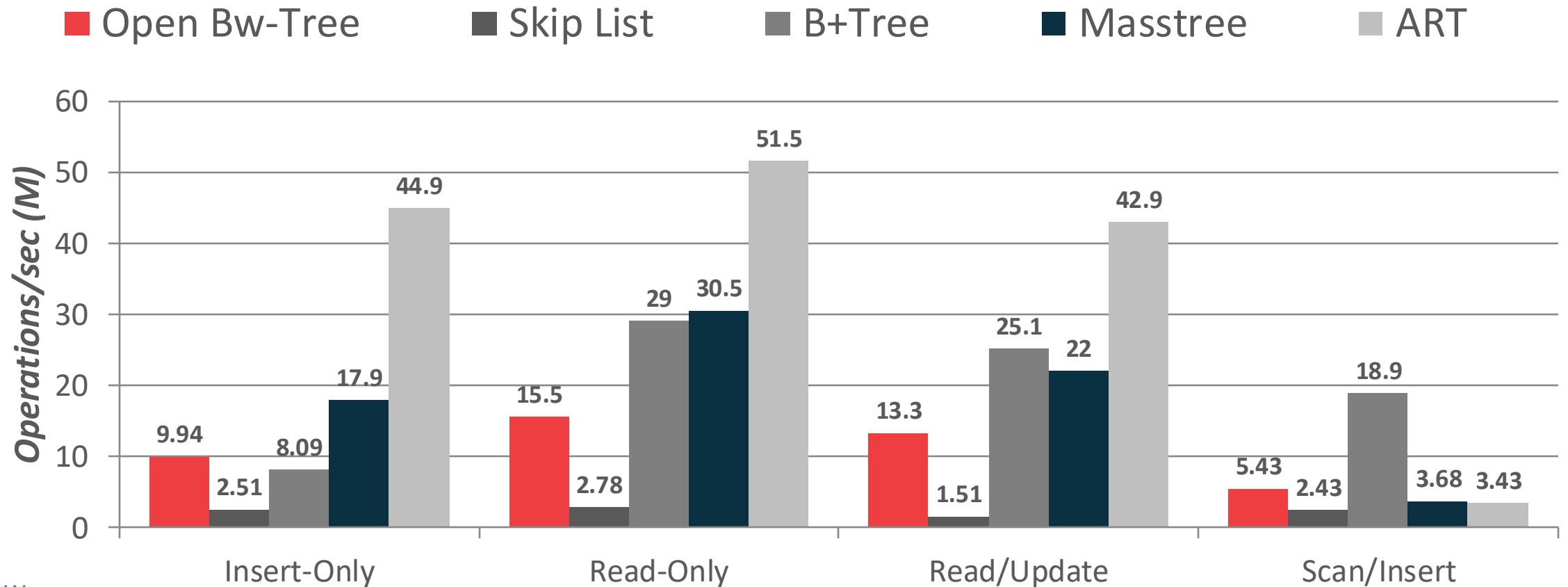
Masstree



- Instead of using different layouts for each trie node based on its size, use an entire B+Tree.
 - Each B+tree represents 8-byte span.
 - Optimized for long keys.
 - Uses a latching protocol that is similar to versioned latches.
- Part of the [Harvard Silo](#) project.

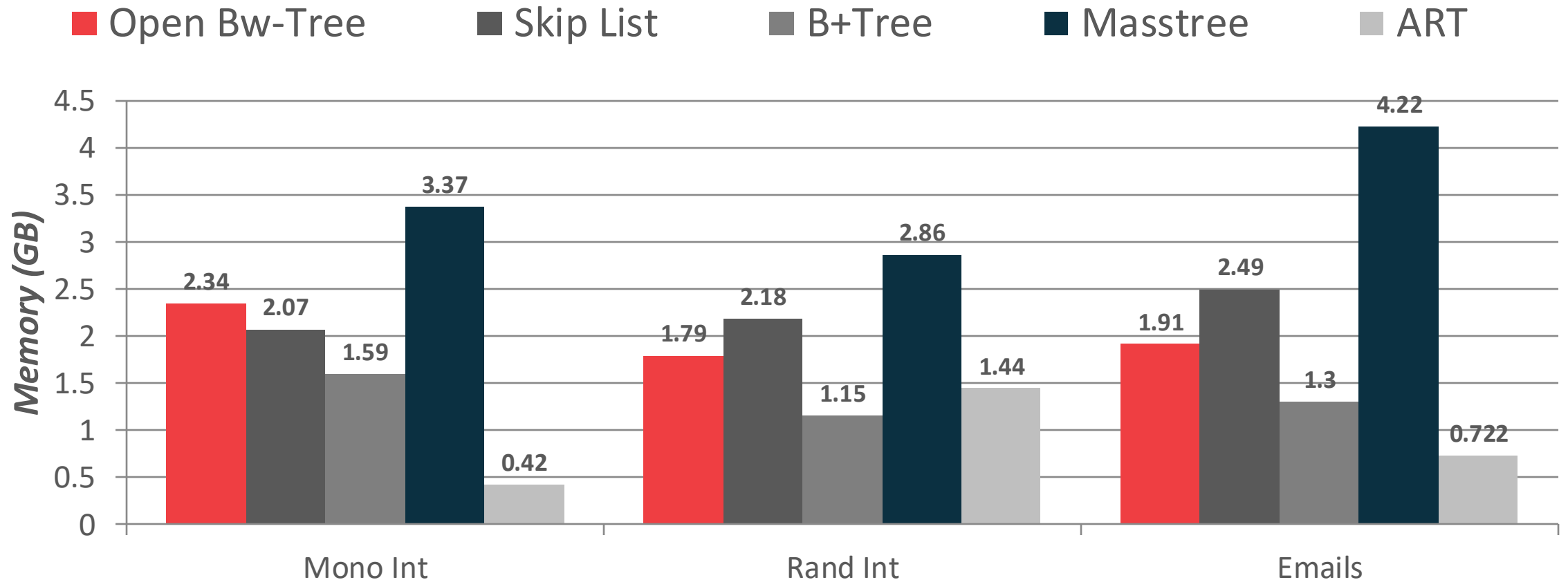
IN-MEMORY INDEXES

*Processor: 1 socket, 10 cores w/ 2×HT
Workload: 50m Random Integer Keys (64-bit)*



IN-MEMORY INDEXES

Processor: 1 socket, 10 cores w/ 2xHT
Workload: 50m Keys



PARTING THOUGHTS

- B+ trees are the go to in-memory indexing data structures.
- Radix trees have interesting properties, but a well-written B+tree is still a solid design choice.
- Skip lists are amazing if you don't want to implement self balancing binary trees

Next class

- Concurrency control

Make sure to read the related papers from the reading list