Modern OLTP Indexes (Part 2)

Recap

Versioned Latch Coupling

- Optimistic coupling scheme where writers are **not** blocked on readers.
- Provides the benefits of optimistic coupling without wasting too much work.
- Every latch has a **version counter**.
- Writers traverse down the tree like a reader
  - Acquire latch in target node to block other writers.
  - Increment version counter before releasing latch.
  - Writer thread increments version counter and acquires latch in a single **compare-and-swap** instruction.
- Reference

Bw-Tree

- Latch-free B+Tree index built for the Microsoft Hekaton project.
- **Key Idea 1: Delta Updates**
  - No in-place updates.
  - Reduces cache invalidation.
- **Key Idea 2: Mapping Table**
  - Allows for CaS of physical locations of pages.
- Reference

Today's Agenda

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

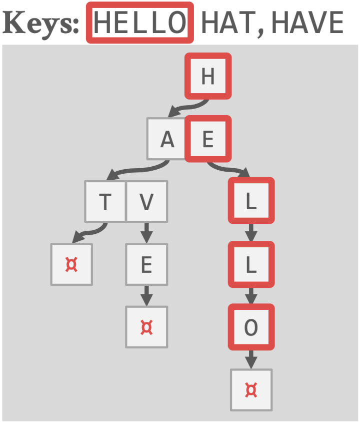
Trie Index

Observation

- The inner node keys in a B+Tree cannot tell you whether a key exists in the index.
- You must always traverse to the leaf node.
- This means that you could have (at least) one buffer pool page miss per level in the tree just to find out a key does not exist.

Trie Index

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



Properties

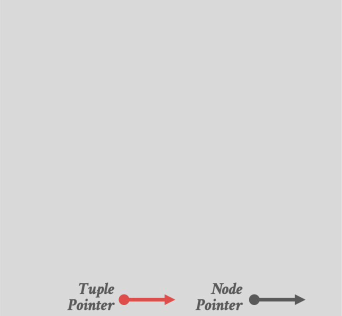
- Shape only depends on key space and lengths.
  - Does not depend on existing keys or insertion order.
  - Does not require rebalancing operations.
- All operations have **O(k)** complexity where **k** is the length of the key.
  - The path to a leaf node represents the key of the leaf
  - Keys are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

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.

Key Span

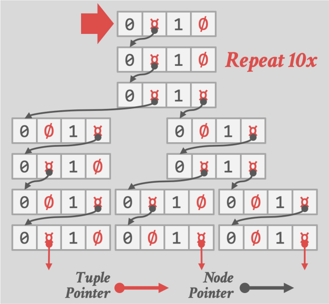
1-bit Span Trie




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

Key Span

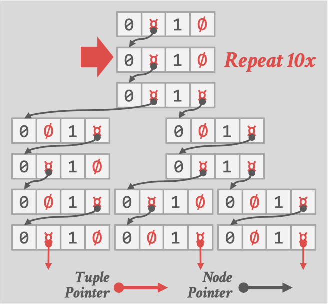
1-bit Span Trie




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

Key Span

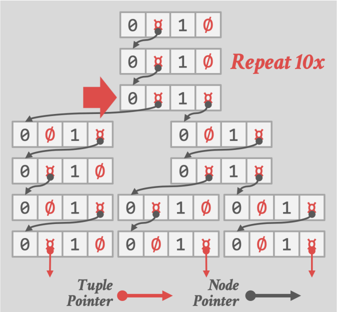
1-bit Span Trie




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

Key Span

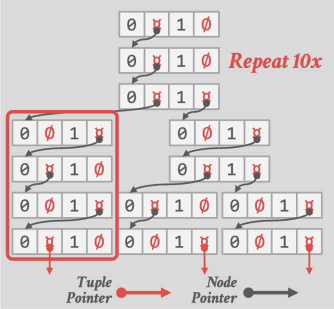
1-bit Span Trie




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

Key Span

1-bit Span Trie

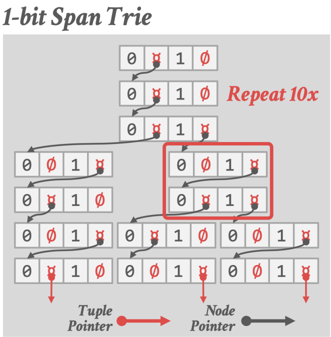


K10→ 00000000 00001010

K25→ 00000000 00011001

K31→ 00000000 00011111

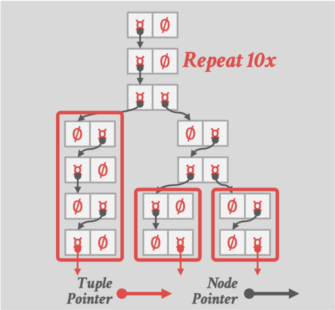
Key Span



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

Key Span

1-bit Span Trie

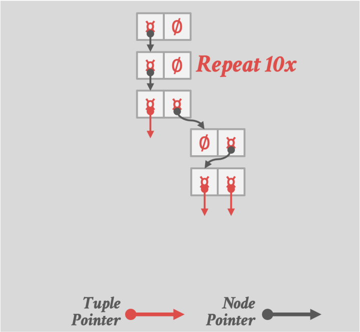



```
K10→ 00000000 00001010
K25→ 00000000 00011001
K31→ 00000000 00011111
```

Radix Tree

- Omit all nodes with <u>**only**</u> a single child.
  - *a.k.a.*, <u>**Patricia Tree**</u>.
- Can produce false positives
- So the DBMS always checks the original tuple to see whether a key matches.

1-bit Span Radix Tree



Trie Variants

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

Judy Arrays

Judy Arrays

- Variant of a 256-way radix tree (since a byte is 8 bits)
- **Goal:** Minimize the amount of cache misses per lookup
- 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.
- Reference

Judy Arrays

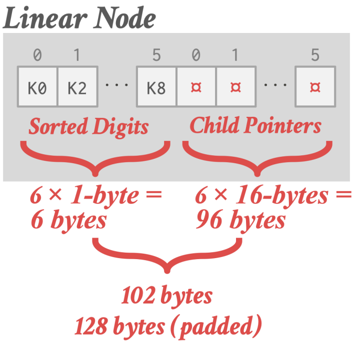
- Do not store meta-data about node in its header.
  - This could lead to additional cache misses.
  - Instead store meta-data in the pointer to that node.
- Pack meta-data about a node in 128-bit **fat pointers** stored in its parent node.
  - Node Type
  - Population Count
  - Child Key Prefix / Value (if only one child below)
  - 64-bit Child Pointer
- Reference

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 (*i.e.*, small number of digits at a level)
  - **Bitmap Node:** Typical Populations
  - **Uncompressed Node:** Dense Population

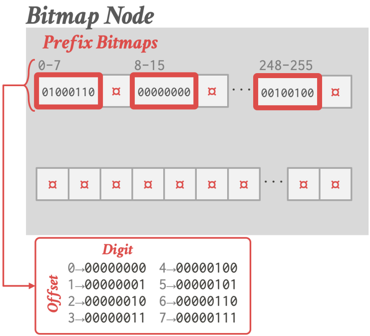
Linear Nodes

- 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.
- Can do a linear scan on sorted digits to find a match.



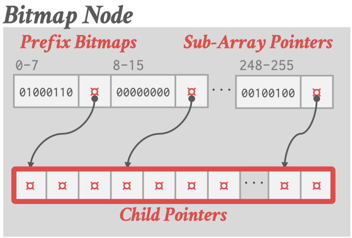
Bitmap Nodes

- 256-bit map to mark whether a prefix (*i.e.*, digit) is present in node.
- Bitmap is divided into eight one-byte chunks
- Each chunk has a pointer to a sub-array with pointers to child nodes.



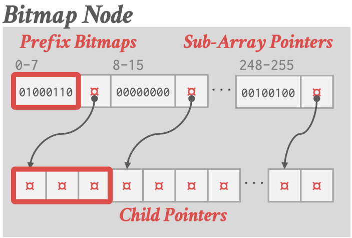
Bitmap Nodes

- To look up a digit (*e.g.*, "1")
- Check at offset 1 in prefix bitmap
- Count the number of 1s that came before offset
- Position to jump into the chunk's sub-array



Bitmap Nodes

- There is a maximum size for the child pointer array
- Although we could present 256 digits in the prefix bitmap, we don't have enough space to store pointers for all of them



Adaptive Radix Tree (ART)

Adaptive Radix Tree (ART)

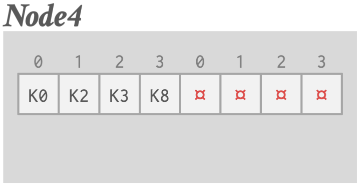
- Developed for TUM's HyPer DBMS in 2013.
- 256-way radix tree that supports different node types based on its population.
  - Stores meta-data about each node in its header.
- Reference

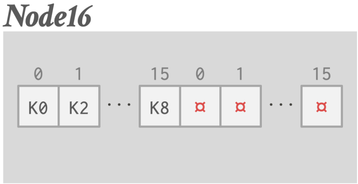
ART vs. JUDY

- **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: Value Type**
  - 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.

Inner Node Types

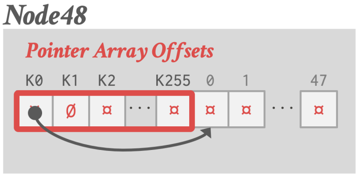
- Store only the 8-bit digits that exist at a given node in a sorted array.
- The offset in sorted digit array corresponds to offset in value array.
- Pack in multiple digits into a single node to improve cache locality.
- First two node types support a small number of digits at that node.
- Use SIMD to quickly find a matching digit per node.





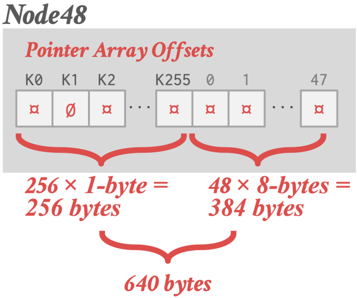
Inner Node Types

- 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.



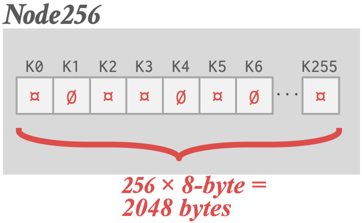
Inner Node Types

- 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.



Inner Node Types

- 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.

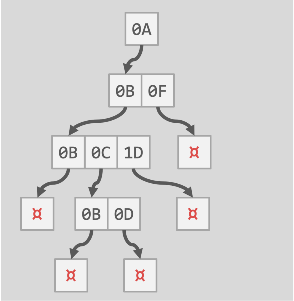


Binary Comparable Keys

- 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.

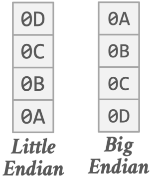
Binary Comparable Keys

8-bit Span Radix Tree



Int Key: 168496141

Hex Key: 0A 0B 0C 0D

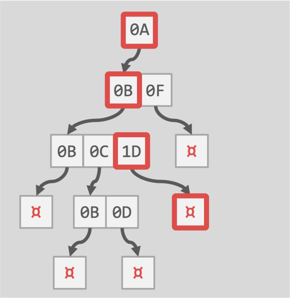


Find 658205

Hex 0A 0B 1D

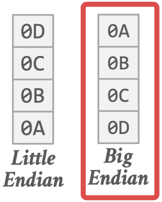
Binary Comparable Keys

8-bit Span Radix Tree



Int Key: 168496141

Hex Key: 0A 0B 0C 0D



Find 658205

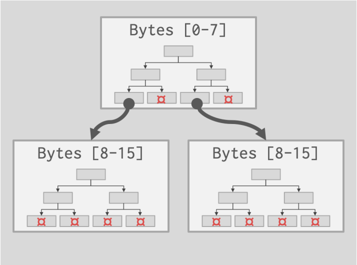
Hex 0A 0B 1D

MassTree

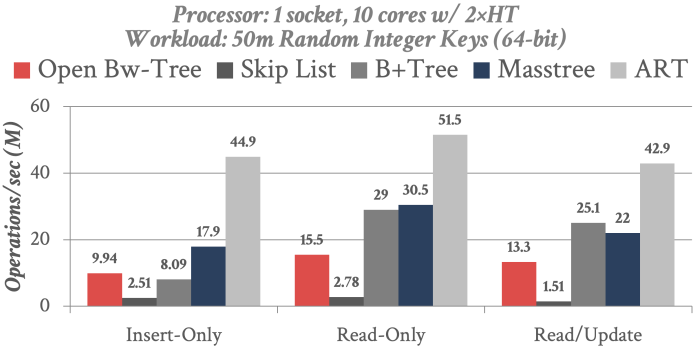
Masstree

- Instead of using different layouts for each trie node based on its size, use an entire B+Tree.
- Part of the Harvard Silo project.
  - Each B+tree represents 8-byte span.
  - Optimized for long keys (*e.g.*, URLs).
  - Uses a latching protocol that is similar to versioned latches.
  - In any trie node, you can have pointers to tuples in the leaf nodes of the B+tree
- Reference

Masstree

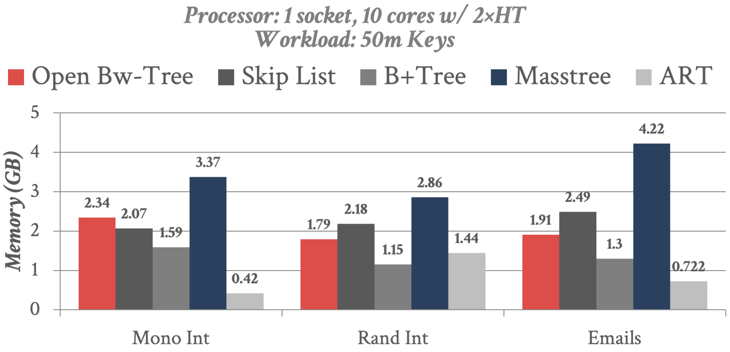


In-Memory Indexes: Performance



Source

In-Memory Indexes: Performance



Source

Conclusion

Conclusion

- Bw-Tree vs ART.
- Radix trees have interesting properties, but a well-written B+tree is still a solid design choice.
- Next Class
  - Executing a query