OLTP Indexes (Trie Data Structures)
TODAY’S AGENDA

Latches
B+Trees
Judy Array
ART
Masstree
LATCH IMPLEMENTATION GOALS

Small memory footprint.

Fast execution path when no contention.
Deschedule thread when it has been waiting for too long to avoid burning cycles.

Each latch should not have to implement their own queue to track waiting threads.

Source: Filip Pizlo
LATCH IMPLEMENTATION GOALS

Small memory footprint.

Fast execution

I repeat: do not use spinlocks in user space, unless you actually know what you're doing. And be aware that the likelihood that you know what you are doing is basically nil.

Source: Filip Pizlo
LATCH IMPLEMENTATIONS

Test-and-Set Spinlock
Blocking OS Mutex
Adaptive Spinlock
Queue-based Spinlock
Reader-Writer Locks
LATCH IMPLEMENTATIONS

Choice #1: Test-and-Set Spinlock (TaS)
→ Very efficient (single instruction to lock/unlock)
→ Non-scalable, not cache friendly, not OS friendly.
→ Example: std::atomic<T>

```
std::atomic_flag latch;
:
while (latch.test_and_set(…)) {
    // Yield? Abort? Retry?
}
```
LATCH IMPLEMENTATIONS

Choice #1: Test-and-Set Spinlock (TaS)
→ Very efficient (single instruction to lock/unlock)
→ Non-scalable, not cache friendly, not OS friendly.
→ Example: \texttt{std::atomic<T>}

\begin{verbatim}
std::atomic_flag latch;
:
while (latch.test_and_set(...)) {
  // Yield? Abort? Retry?
}
\end{verbatim}
LATCH IMPLEMENTATIONS

Choice #2: Blocking OS Mutex
→ Simple to use
→ Non-scalable (about 25ns per lock/unlock invocation)
→ Example: `std::mutex`

```cpp
std::mutex m;
:
m.lock();
// Do something special...
m.unlock();
```
LATCH IMPLEMENTATIONS

Choice #2: Blocking OS Mutex

→ Simple to use
→ Non-scalable (about 25ns per lock/unlock invocation)
→ Example: `std::mutex`

```cpp
std::mutex m;

m.lock(); // Do something special...

m.unlock();
```

Example:
```cpp
pthread_mutex_t futex;
```
LATCH IMPLEMENTATIONS

Choice #2: Blocking OS Mutex
→ Simple to use
→ Non-scalable (about 25ns per lock/unlock invocation)
→ Example: `std::mutex`

```cpp
std::mutex m;
pthread_mutex_t futex;
m.lock();
// Do something special...
m.unlock();
```

OS Latch
Userspace Latch
LATCH IMPLEMENTATIONS

Choice #2: Blocking OS Mutex

→ Simple to use
→ Non-scalable (about 25ns per lock/unlock invocation)
→ Example: `std::mutex`

```
std::mutex m; // Create a mutex
pthread_mutex_t futex; // Use futex for synchronization
m.lock(); // Lock the mutex
// Do something special...
// Example: perform database operations
m.unlock(); // Unlock the mutex
```

- `OS Latch`
- `Userspace Latch`
LATCH IMPLEMENTATIONS

Choice #3: Adaptive Spinlock
→ Thread spins on a userspace lock for a brief time.
→ If they cannot acquire the lock, they then get descheduled and stored in a global "parking lot".
→ Threads check to see whether other threads are "parked" before spinning and then park themselves.
→ Example: Apple's WTF::ParkingLot
Choice #4: Queue-based Spinlock (MCS)
→ More efficient than mutex, better cache locality
→ Non-trivial memory management
→ Example: `std::atomic<Latch*>`
Choice #4: Queue-based Spinlock (MCS)
→ More efficient than mutex, better cache locality
→ Non-trivial memory management
→ Example: `std::atomic<Latch*>`

Mellor-Crummey and Scott
LATCH IMPLEMENTATIONS

Choice #4: Queue-based Spinlock (MCS)
→ More efficient than mutex, better cache locality
→ Non-trivial memory management
→ Example: std::atomic<Latch*>
LATCH IMPLEMENTATIONS

Choice #4: Queue-based Spinlock (MCS)
→ More efficient than mutex, better cache locality
→ Non-trivial memory management
→ Example: `std::atomic<Latch*>`

Mellor-Crummey and Scott

Base Latch

next

CPU1 Latch

next

CPU1
LATCH IMPLEMENTATIONS

Choice #4: Queue-based Spinlock (MCS)
→ More efficient than mutex, better cache locality
→ Non-trivial memory management
→ Example: `std::atomic<Latch*>`

Mellor-Crummey and Scott
LATCH IMPLEMENTATIONS

Choice #4: Queue-based Spinlock (MCS)
→ More efficient than mutex, better cache locality
→ Non-trivial memory management
→ Example: `std::atomic<Latch*>`

Mellor-Crummey and Scott
LATCH IMPLEMENTATIONS

Choice #4: Queue-based Spinlock (MCS)
→ More efficient than mutex, better cache locality
→ Non-trivial memory management
→ Example: `std::atomic<Latch*>`

Mellor-Crummey and Scott

Base Latch → CPU1 Latch → CPU2 Latch

CPU1

CPU2
Choice #4: Queue-based Spinlock (MCS)
→ More efficient than mutex, better cache locality
→ Non-trivial memory management
→ Example: `std::atomic<Latch*>`
Choice #4: Queue-based Spinlock (MCS)
→ More efficient than mutex, better cache locality
→ Non-trivial memory management
→ Example: `std::atomic<Latch*>`

Mellor-Crummey and Scott
Choice #4: Queue-based Spinlock (MCS)
→ More efficient than mutex, better cache locality
→ Non-trivial memory management
→ Example: `std::atomic<Latch*>`

Mellor-Crummey and Scott
LATCH IMPLEMENTATIONS

Choice #5: Reader-Writer Locks
→ Allows for concurrent readers.
→ Must manage read/write queues to avoid starvation.
→ Can be implemented on top of spinlocks.
**LATCH IMPLEMENTATIONS**

**Choice #5: Reader-Writer Locks**

→ Allows for concurrent readers.
→ Must manage read/write queues to avoid starvation.
→ Can be implemented on top of spinlocks.
LATCH IMPLEMENTATIONS

Choice #5: Reader-Writer Locks
→ Allows for concurrent readers.
→ Must manage read/write queues to avoid starvation.
→ Can be implemented on top of spinlocks.
LATCH IMPLEMENTATIONS

Choice #5: Reader-Writer Locks
→ Allows for concurrent readers.
→ Must manage read/write queues to avoid starvation.
→ Can be implemented on top of spinlocks.
LATCH IMPLEMENTATIONS

Choice #5: Reader-Writer Locks
→ Allows for concurrent readers.
→ Must manage read/write queues to avoid starvation.
→ Can be implemented on top of spinlocks.

Latch
read
write

Latch

= 0
= 0
= 2
= 0
= 0
= 0
Choice #5: Reader-Writer Locks
→ Allows for concurrent readers.
→ Must manage read/write queues to avoid starvation.
→ Can be implemented on top of spinlocks.
LATCH IMPLEMENTATIONS

Choice #5: Reader-Writer Locks
→ Allows for concurrent readers.
→ Must manage read/write queues to avoid starvation.
→ Can be implemented on top of spinlocks.
LATCH IMPLEMENTATIONS

Choice #5: Reader-Writer Locks
→ Allows for concurrent readers.
→ Must manage read/write queues to avoid starvation.
→ Can be implemented on top of spinlocks.
LATCH IMPLEMENTATIONS

Choice #5: Reader-Writer Locks
→ Allows for concurrent readers.
→ Must manage read/write queues to avoid starvation.
→ Can be implemented on top of spinlocks.
A **B+Tree** is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in $O(\log n)$.

→ Generalization of a binary search tree in that a node can have more than two children.

→ Optimized for systems that read and write large blocks of data.
LATCH CRABBING / COUPLING

Acquire and release latches on B+Tree nodes when traversing the data structure.

A thread can release latch on a parent node if its child node considered **safe**.
→ Any node that won’t split or merge when updated.
→ Not full (on insertion)
→ More than half-full (on deletion)
LATCH CRABBING

**Search**: Start at root and go down; repeatedly,
→ Acquire read (R) latch on child
→ Then unlock the parent node.

**Insert/Delete**: Start at root and go down, obtaining write (W) latches as needed.
Once child is locked, check if it is safe:
→ If child is safe, release all locks on ancestors.
EXAMPLE #1: SEARCH 23

A

B

10

D

12

E

23

F

38 44

C

G

6
EXAMPLE #1: SEARCH 23
EXAMPLE #1: SEARCH 23

We can release the latch on A as soon as we acquire the latch for C.
EXAMPLE #1: SEARCH 23

We can release the latch on A as soon as we acquire the latch for C.
EXAMPLE #1: SEARCH 23

We can release the latch on A as soon as we acquire the latch for C.
EXAMPLE #2: DELETE 44
EXAMPLE #2: DELETE 44

We may need to coalesce C, so we can’t release the latch on A.
EXAMPLE #2: DELETE 44

We may need to coalesce C, so we can’t release the latch on A.

G will not merge with F, so we can release latches on A and C.
EXAMPLE #2: DELETE 44

We may need to coalesce C, so we can’t release the latch on A.

G will not merge with F, so we can release latches on A and C.
EXAMPLE #3: INSERT 40

Diagram showing a data structure with nodes labeled A, B, C, D, E, F, and G. Nodes contain values such as 10, 20, 35, 6, 12, 23, 38, and 44. The diagram illustrates the process of inserting a value into the structure.
EXAMPLE #3: INSERT 40

C has room if its child has to split, so we can release the latch on A.
EXAMPLE #3: INSERT 40

C has room if its child has to split, so we can release the latch on A.
EXAMPLE #3: INSERT 40

C has room if its child has to split, so we can release the latch on A.

G must split, so we can’t release the latch on C.
EXAMPLE #3: INSERT 40

C has room if its child has to split, so we can release the latch on A.

G must split, so we can’t release the latch on C.
EXAMPLE #3: INSERT 40

C has room if its child has to split, so we can release the latch on A.

G must split, so we can’t release the latch on C.
The basic latch crabbing algorithm always takes a write latch on the root for any update.
   → This makes the index essentially single threaded.

A better approach is to optimistically assume that the target leaf node is safe.
   → Take R latches as you traverse the tree to reach it and verify.
   → If leaf is not safe, then do previous algorithm.
EXAMPLE #4: DELETE 44

```
10  B  12
D  23  38  44
G

A

20

C

35

R
```
EXAMPLE #4: DELETE 44

We assume that C is safe, so we can release the latch on A.
We assume that C is safe, so we can release the latch on A.
Acquire an exclusive latch on G.
EXAMPLE #4: DELETE 44

We assume that C is safe, so we can release the latch on A.

Acquire an exclusive latch on G.
VERSIONED LATCH COUPLING

Optimistic crabbing scheme where writers are not blocked on readers.

Every node now has a version number (counter).
→ Writers increment counter when they acquire latch.
→ Readers proceed if a node’s latch is available but then do not acquire it.
→ It then checks whether the latch’s counter has changed from when it checked the latch.

Relies on epoch GC to ensure pointers are valid.
VERSIONED LATCHES: SEARCH 44
VERSIONED LATCHES: SEARCH 44

@A A: Read v3
VERSIONED LATCHES: SEARCH 44

@A A: Read v3
A: Examine Node

@B
VERSIONED LATCHES: SEARCH 44

A: Read v3
A: Examine Node

B: Read v5

@A

@B
VERSIONED LATCHES: SEARCH 44

@A  
A: Read v3  
A: Examine Node

B: Read v5

@B  
A: Recheck v3
VERSIONED LATCHES: SEARCH 44

A: Read v3
A: Examine Node

B: Read v5
B: Examine Node

@A

@B
VERSIONED LATCHES: SEARCH 44

A: Read v3
A: Examine Node

B: Read v5
B: Examine Node

C: Read v9

@A

@B

@C
VERSIONED LATCHES: SEARCH 44

A: Read v3
A: Examine Node

B: Read v5

@A

B: Examine Node

C: Read v9
C: Examine Node

@B

B: Recheck v3
B: Recheck v5

@C

Diagram:

- A: 20
- B: 10
- D: 6
- E: 23
- F: 38
- C: 35
- G: 44

Node versions:

- v3
- v5
- v4
- v9
- v6

Process:

1. A: Read v3
2. A: Examine Node
3. B: Read v5
4. @A
5. B: Examine Node
6. @B
7. C: Read v9
8. C: Examine Node
9. @C
10. B: Recheck v3
11. B: Recheck v5
12. @B

VERSIONED LATCHES: SEARCH 44

A: Read v3
A: Examine Node

B: Read v5
A: Recheck v3
B: Examine Node

C: Read v9
B: Recheck v5
C: Examine Node
VERSIONED LATCHES: SEARCH 44

A: Read v3
A: Examine Node

B: Read v5

B: Examine Node

C: Read v9

@A

@B

@C
VERSIONED LATCHES: SEARCH 44

@A
A: Read v3
A: Examine Node

@B
B: Read v5
A: Recheck v3
B: Examine Node

@C
C: Read v9

Diagram shows nodes labeled with versions and locking operations.
VERSIONED LATCHES: SEARCH 44

A: Read v3
A: Examine Node

B: Read v5
B: Examine Node

C: Read v9
B: Recheck v5

A: Recheck v3
B: Recheck v5

@A
@B
@C
VERSIONED LATCHES: SEARCH 44

A: Read v3
A: Examine Node
B: Read v5
B: Examine Node
C: Read v9
B: Recheck v5

Diagram:

- A
- B
- C
- D
- E
- F
- G

Nodes:
- v3
- v4
- v5
- v6
- v9

Latches:
- v3
- v5
- v6
- v9

Actions:
- A: Read v3
- A: Examine Node
- B: Read v5
- B: Examine Node
- C: Read v9
- B: Recheck v5

Graph:

- A to B
- B to C
- C to A
- D to E
- E to F
- F to G
- G to D

Locks:
- A
- B
- C

Version Numbers:
- v3
- v5
- v6
- v9
OBSERVATION

The inner node keys in a B+tree cannot tell you whether a key exists in the index. You always must traverse to the leaf node.

This means that you could have (at least) one cache miss per level in the tree.
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

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.
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

Keys: K10, K25, K31

K10→ 00000000 00001010
K25→ 00000000 00011001
K31→ 00000000 00011111
TRIE KEY SPAN

1-bit Span Trie

Keys: K10, K25, K31

K10 → 00000000 00001010
K25 → 00000000 00011001
K31 → 00000000 00011111
TRIE KEY SPAN

1-bit Span Trie

**Keys:** K10, K25, K31

- **K10** → 00000000 00001010
- **K25** → 00000000 00011001
- **K31** → 00000000 00011111
TRIE KEY SPAN

1-bit Span Trie

Keys: K10, K25, K31

K10 → 00000000 00001010
K25 → 00000000 00011001
K31 → 00000000 00011111
TRIE KEY SPAN

1-bit Span Trie

Keys: K10, K25, K31

K10 → 00000000 00001010
K25 → 00000000 00011001
K31 → 00000000 00011111
**TRIE KEY SPAN**

1-bit Span Trie

**Keys:** K10, K25, K31

K10→ 00000000 00001010
K25→ 00000000 00011001
K31→ 00000000 00011111
TRIE KEY SPAN

1-bit Span Trie

Keys: K10, K25, K31

K10→ 00000000 00001010
K25→ 00000000 00011001
K31→ 00000000 00011111
TRIE KEY SPAN

1-bit Span Trie

Keys: K10, K25, K31

K10 → 00000000 00001010
K25 → 00000000 00011001
K31 → 00000000 00011111
TRIE KEY SPAN

1-bit Span Trie

Keys: K10, K25, K31

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

→ 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: LINEAR NODES

Linear Node

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>5</th>
<th>0</th>
<th>1</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>K0</strong></td>
<td><strong>K2</strong></td>
<td>⋯</td>
<td><strong>K8</strong></td>
<td>⋯</td>
<td></td>
</tr>
</tbody>
</table>

*Sorted Digits*

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: 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.
JUDY ARRAYS: 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.

**Linear Node**

<table>
<thead>
<tr>
<th>Sorted Digits</th>
<th>Child Pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td>K0 1 5</td>
<td>K2 K8</td>
</tr>
<tr>
<td>K8</td>
<td>□ □</td>
</tr>
<tr>
<td>□</td>
<td>□</td>
</tr>
</tbody>
</table>

6 × 1-byte = 6 bytes
6 × 16-bytes = 96 bytes
102 bytes
128 bytes (padded)
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.
JUDY ARRAYS: BITMAP NODES

**Bitmap Node**

**Prefix Bitmaps**

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

- 0→01000110
- 8→00000000
- 248→00100100

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.
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.
JUDY ARRAYS: BITMAP NODES

Bitmap Node

Prefix Bitmaps

Sub-Array Pointers

Child Pointers

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.
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)

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.
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.
ART: INNER NODE TYPES (1)

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.
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.
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 (2)

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.
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.

\[256 \times 1\text{-byte} = 256\text{ bytes}\]
\[48 \times 8\text{-bytes} = 384\text{ bytes}\]

640 bytes
ART: INNER NODE TYPES (3)

Node256

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: INNER NODE TYPES (3)

Node256

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.

256 × 8-byte = 2048 bytes
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
ART: BINARY COMPARABLE KEYS

8-bit Span Radix Tree

Int Key: 168496141

Hex Key: 0A 0B 0C 0D

Find: 658205

Hex: 0A 0B 1D
ART: BINARY COMPARABLE KEYS

8-bit Span Radix Tree

Int Key: 168496141

Hex Key: 0A 0B 0C 0D

Find: 658205

Hex: 0A 0B 1D
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)*

<table>
<thead>
<tr>
<th>Operation Type</th>
<th>Open Bw-Tree</th>
<th>Skip List</th>
<th>B+Tree</th>
<th>Masstree</th>
<th>ART</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert-Only</td>
<td>9.94</td>
<td>44.9</td>
<td>17.9</td>
<td>15.5</td>
<td>2.51</td>
</tr>
<tr>
<td>Read-Only</td>
<td>2.51</td>
<td>29</td>
<td>30.5</td>
<td>22</td>
<td>5.43</td>
</tr>
<tr>
<td>Read/Update</td>
<td>13.3</td>
<td>25.1</td>
<td>22</td>
<td>42.9</td>
<td>3.68</td>
</tr>
<tr>
<td>Scan/Insert</td>
<td>5.43</td>
<td>18.9</td>
<td>3.43</td>
<td>3.68</td>
<td>3.43</td>
</tr>
</tbody>
</table>

Source: Ziqi Wang
IN-MEMORY INDEXES

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

Source: Ziqi Wang
PARTING THOUGHTS

Andy was wrong about the Bw-Tree and latch-free indexes.

Radix trees have interesting properties, but a well-written B+tree is still a solid design choice.
NEXT CLASS

System Catalogs
Data Layout
Storage Models