Lecture #07

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

OLTP Indexes (Trie Data Structures)

@Andy_Pavlo // 15-721 // Spring 2020

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 IM

Small memory

Fast execution

By: **Linus Torvalds** (torvalds.delete@this.linux-foundation.org), January 3, 2020 6:05 pm Room: Moderated Discussions

Beastian (no.email.delete@this.aol.com) on January 3, 2020 11:46 am wrote:
> I'm usually on the other side of these primitives when I write code as a consumer of them,
> but it's very interesting to read about the nuances related to their implementations:

The whole post seems to be just wrong, and is measuring something completely different than what the author thinks and claims it is measuring.

First off, spinlocks can only be used if you actually know you're not being scheduled while using them. But the blog post author seems to be implementing his own spinlocks in user space with no regard for whether the lock user might be scheduled or not. And the code used for the claimed "lock not held" timing is complete garbage.

It basically reads the time before releasing the lock, and then it reads it after acquiring the lock again, and claims that the time difference is the time when no lock was held. Which is just inane and pointless and completely wrong.

That's pure garbage. What happens is that

(a) since you're spinning, you're using CPU time

(b) at a random time, the scheduler will schedule you out

(c) that random time might ne just after you ...

ime" you

it's still
e", and
looks at

d to your

n

... and getting nonsensical values, ... rather, you very much can do them like that, and when you do ... all the processes are CPU-bound, and I'm measuring random points of how long the scheduler kept the process in place". ... because what you are measuring is "I have a lot of busywork,

And then you write a blog-post blamings others, not understanding that it's your incorrect code that is garbage, and is giving random garbage values.

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<bool>

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

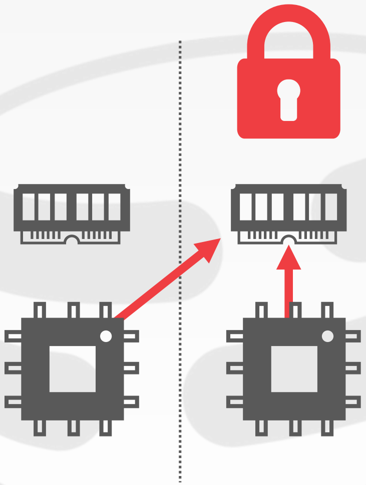
LATCH IMPLEMENTATIONS

Choice #1: Test-and-Set Spinlock (TaS)

→ Very efficient (single instruction to lock/unlock)
→ Non-scalable, not cache friendly, not OS friendly.
→ Example: **std::atomic<T>**

std::atomic<bool>

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



LATCH IMPLEMENTATIONS

Choice #2: Blocking OS Mutex

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

```
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**

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

LATCH IMPLEMENTATIONS

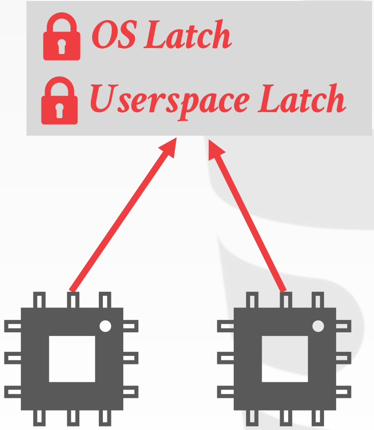
Choice #2: Blocking OS Mutex

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

```
std::mutex m;
  ⋮
m.lock();
// Do something special...
m.unlock();
```

pthread_mutex_t

futex



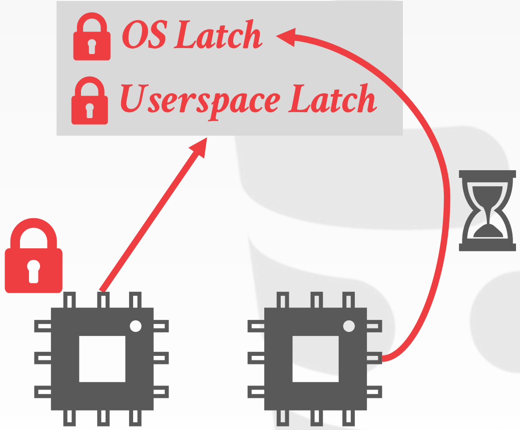
LATCH IMPLEMENTATIONS

Choice #2: Blocking OS Mutex

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

```
std::mutex m;
  ⋮
m.lock();
// Do something special...
m.unlock();
```

pthread_mutex_t → futex



LATCH IMPLEMENTATIONS

Choice #3: Adaptive Spinlock

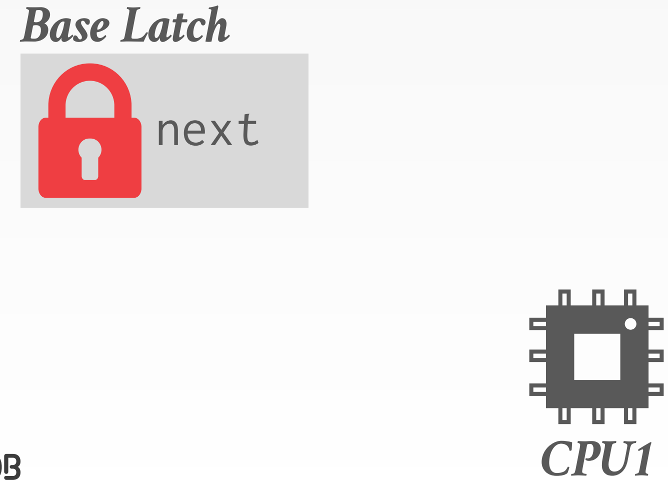
→ Thread spins on a userspace lock for a brief time.
→ If they cannot acquire the lock, they then get descheduled and stored in a global "parking lot".
→ Threads check to see whether other threads are "parked" before spinning and then park themselves.
→ Example: Apple's `WTF::ParkingLot`

LATCH IMPLEMENTATIONS

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*>`**

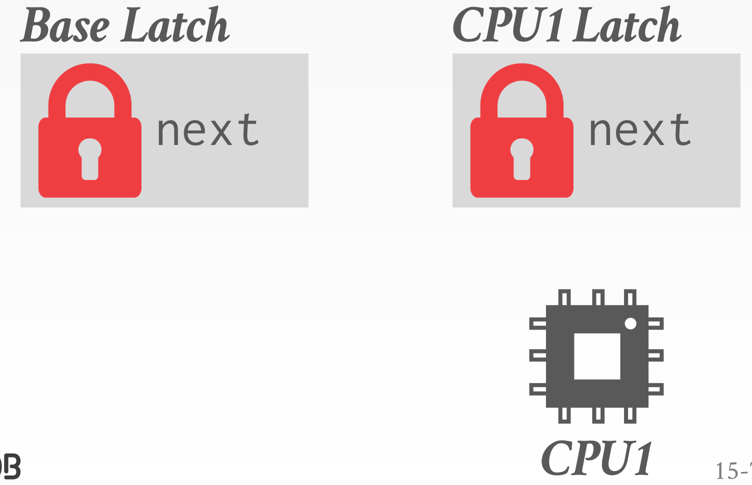


LATCH IMPLEMENTATIONS

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*>`

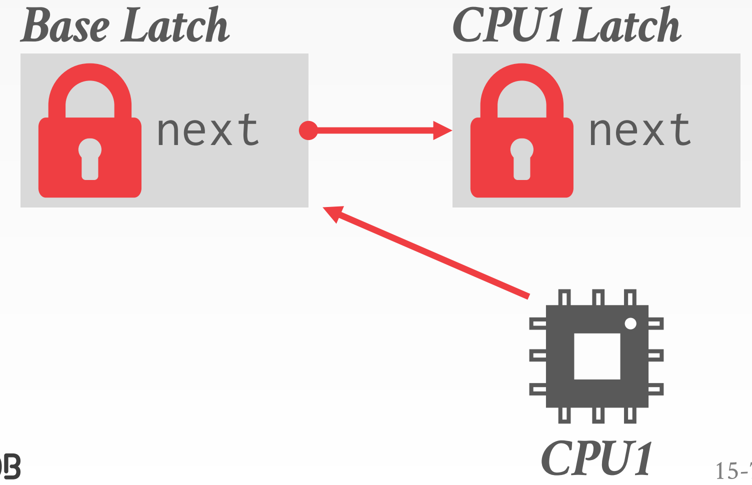


LATCH IMPLEMENTATIONS

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*>`

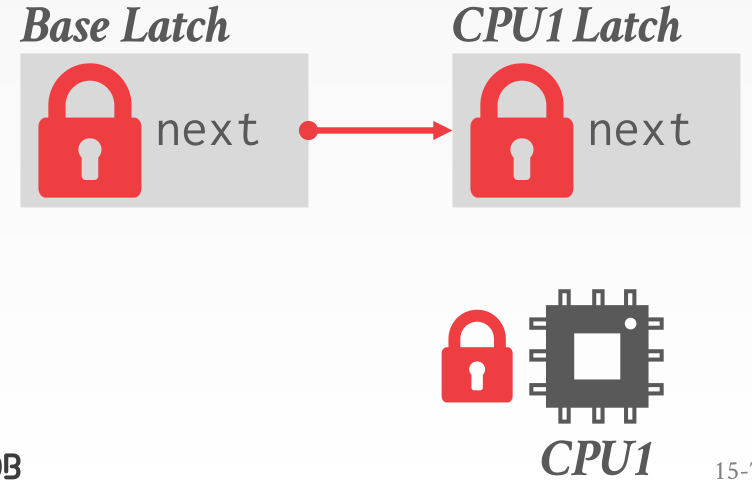


LATCH IMPLEMENTATIONS

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*>`



LATCH IMPLEMENTATIONS

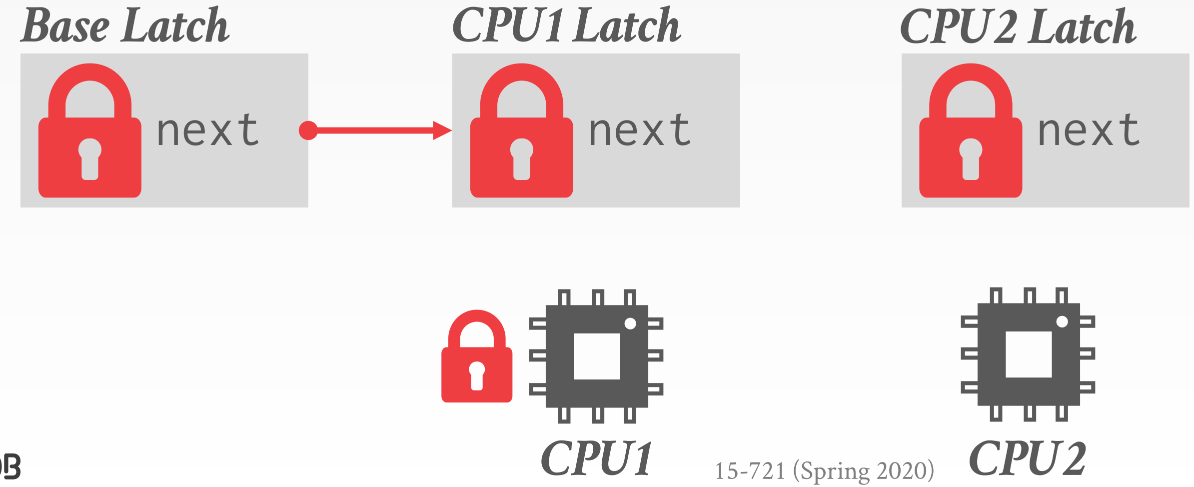
Mellor-Crummey and Scott

Choice #4: Queue-based Spinlock (MCS)

→ More efficient than mutex, better cache locality

→ Non-trivial memory management

→ Example: `std::atomic<Latch*>`

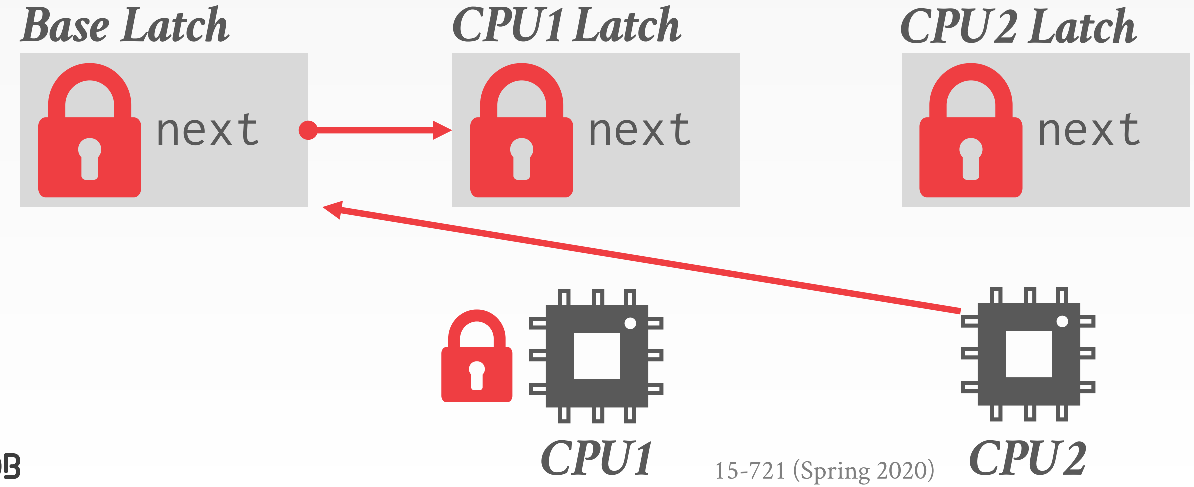


LATCH IMPLEMENTATIONS

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*>`

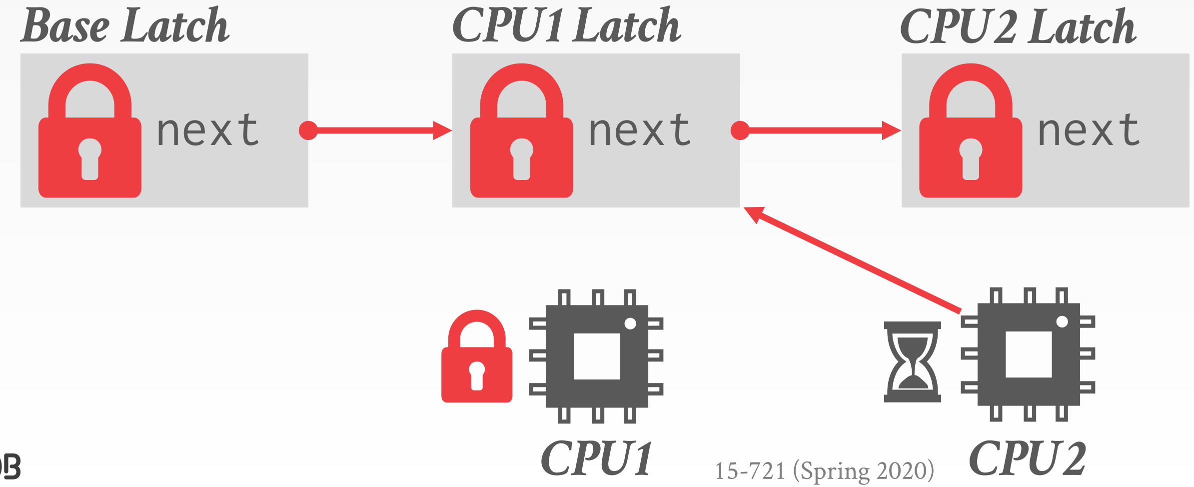


LATCH IMPLEMENTATIONS

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*>`

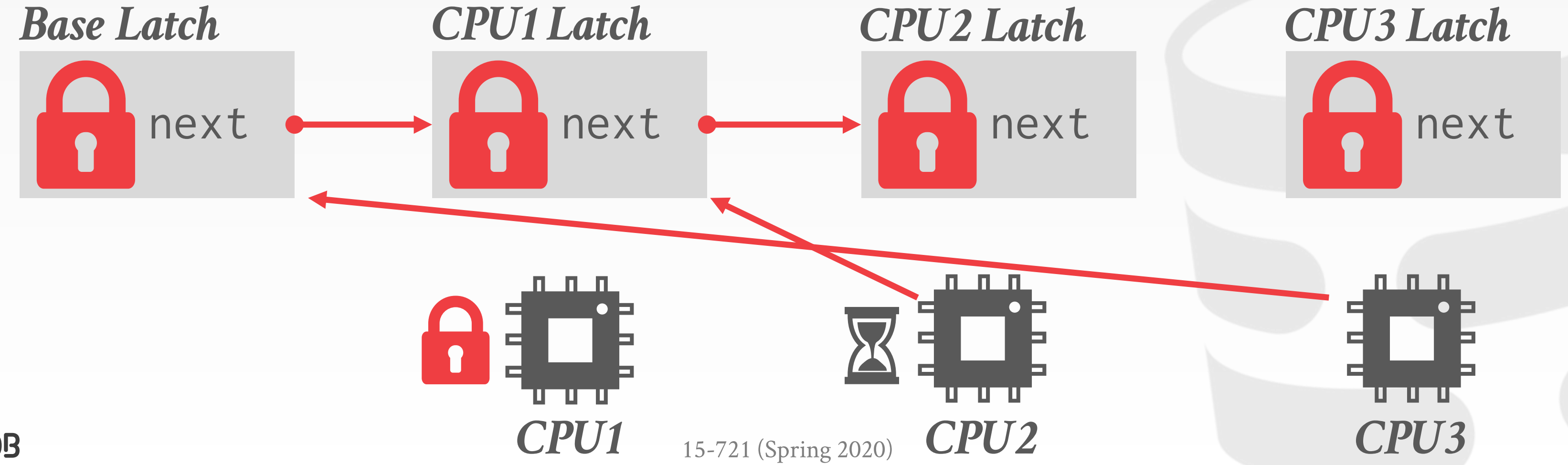


LATCH IMPLEMENTATIONS

Choice #4: Queue-based Spinlock (MCS)

Mellor-Crummey and Scott

→ More efficient than mutex, better cache locality
→ Non-trivial memory management
→ Example: `std::atomic<Latch*>`

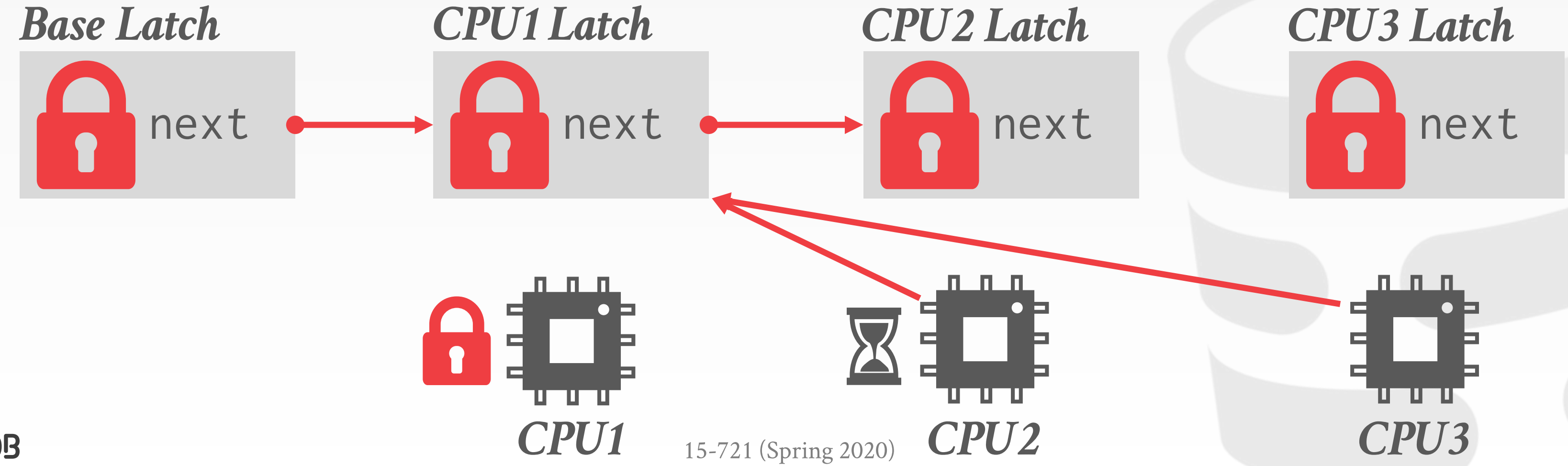


LATCH IMPLEMENTATIONS

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*>`**



LATCH IMPLEMENTATIONS

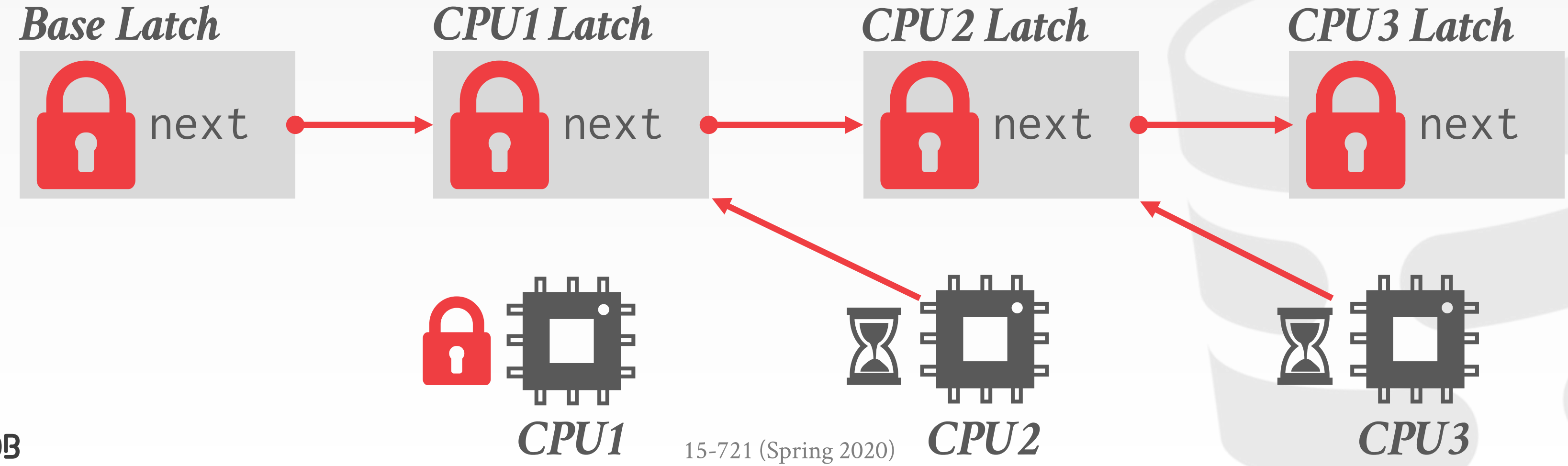
Mellor-Crummey and Scott

Choice #4: Queue-based Spinlock (MCS)

→ More efficient than mutex, better cache locality

→ Non-trivial memory management

→ Example: **`std::atomic<Latch*>`**



LATCH IMPLEMENTATIONS

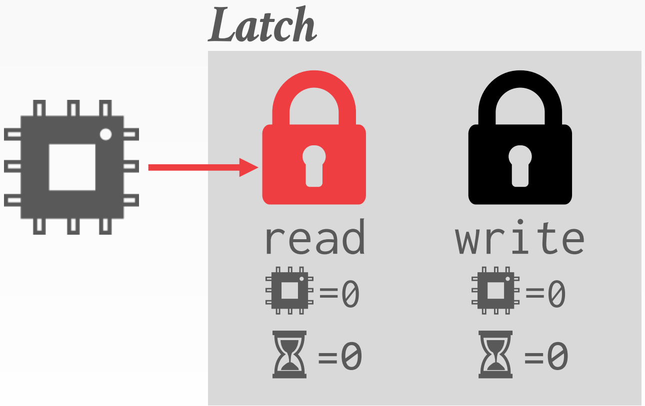
Choice #5: Reader-Writer Locks

→ 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

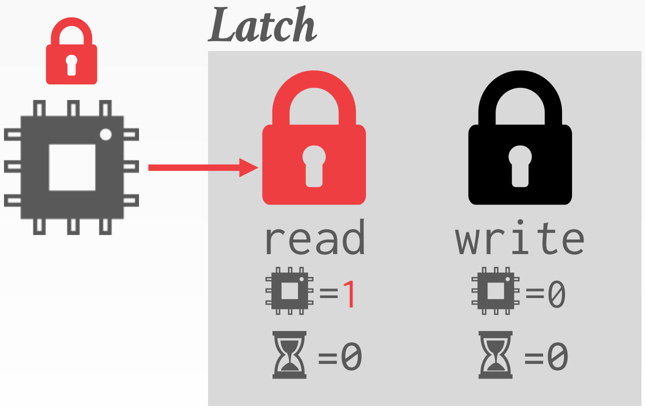
→ Allows for concurrent readers.
→ Must manage read/write queues to avoid starvation.
→ Can be implemented on top of spinlocks.



LATCH IMPLEMENTATIONS

Choice #5: Reader-Writer Locks

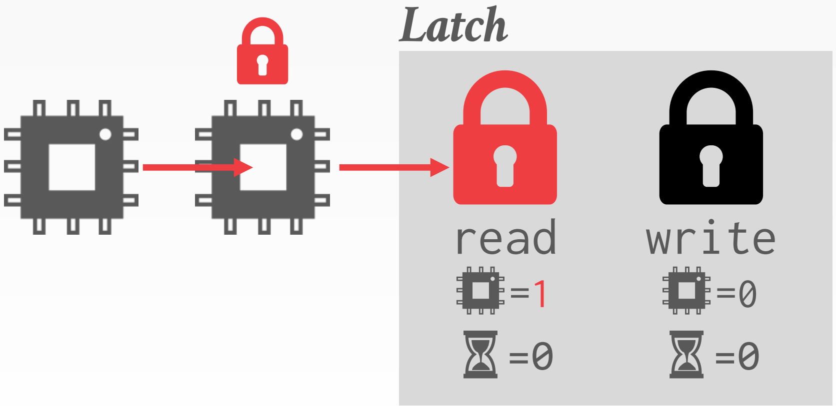
→ Allows for concurrent readers.
→ Must manage read/write queues to avoid starvation.
→ Can be implemented on top of spinlocks.



LATCH IMPLEMENTATIONS

Choice #5: Reader-Writer Locks

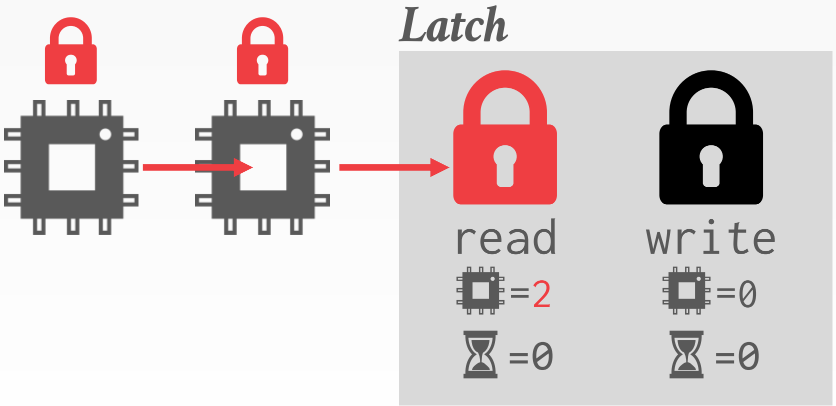
→ Allows for concurrent readers.
→ Must manage read/write queues to avoid starvation.
→ Can be implemented on top of spinlocks.



LATCH IMPLEMENTATIONS

Choice #5: Reader-Writer Locks

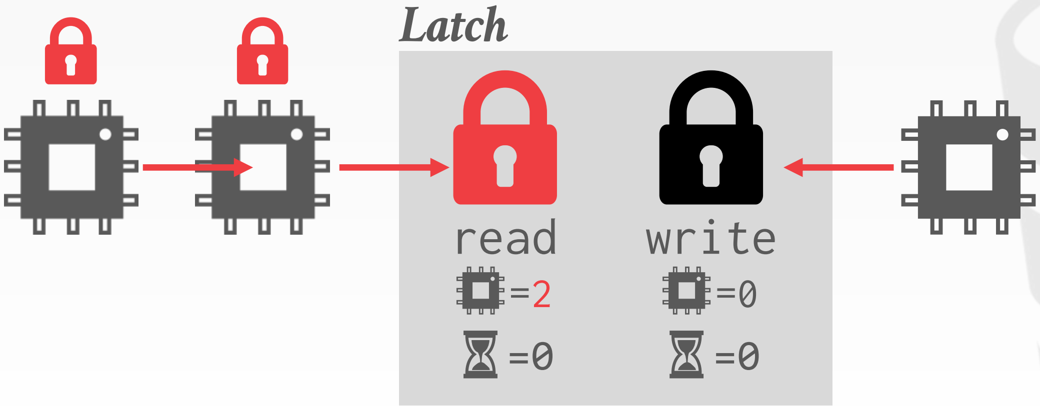
→ Allows for concurrent readers.
→ Must manage read/write queues to avoid starvation.
→ Can be implemented on top of spinlocks.



LATCH IMPLEMENTATIONS

Choice #5: Reader-Writer Locks

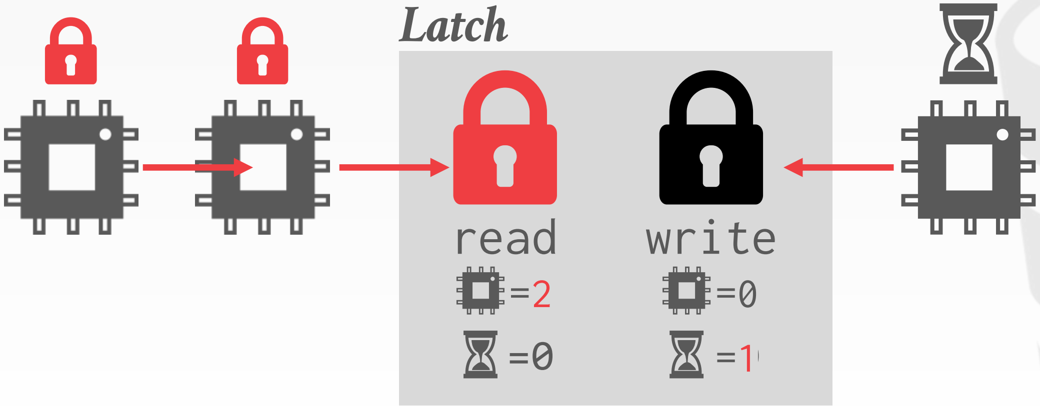
→ Allows for concurrent readers.
→ Must manage read/write queues to avoid starvation.
→ Can be implemented on top of spinlocks.



LATCH IMPLEMENTATIONS

Choice #5: Reader-Writer Locks

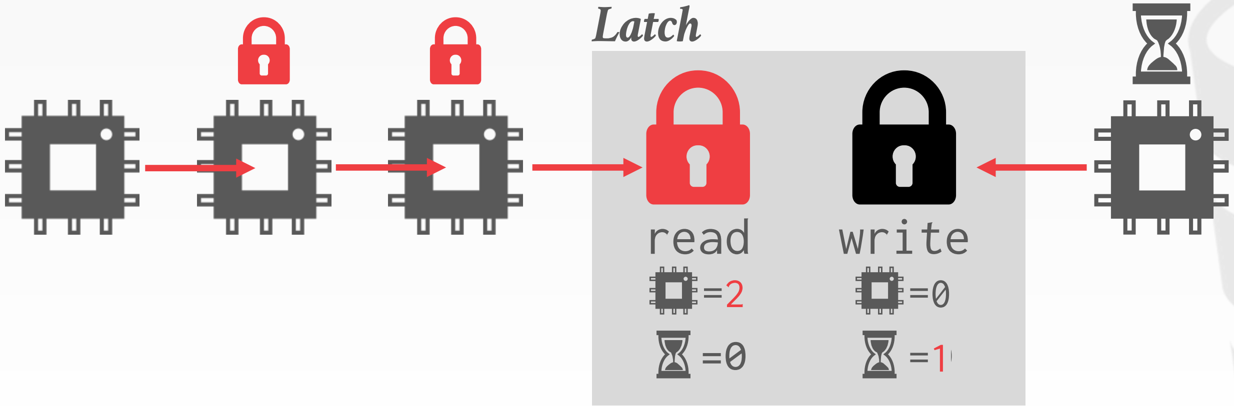
→ Allows for concurrent readers.
→ Must manage read/write queues to avoid starvation.
→ Can be implemented on top of spinlocks.



LATCH IMPLEMENTATIONS

Choice #5: Reader-Writer Locks

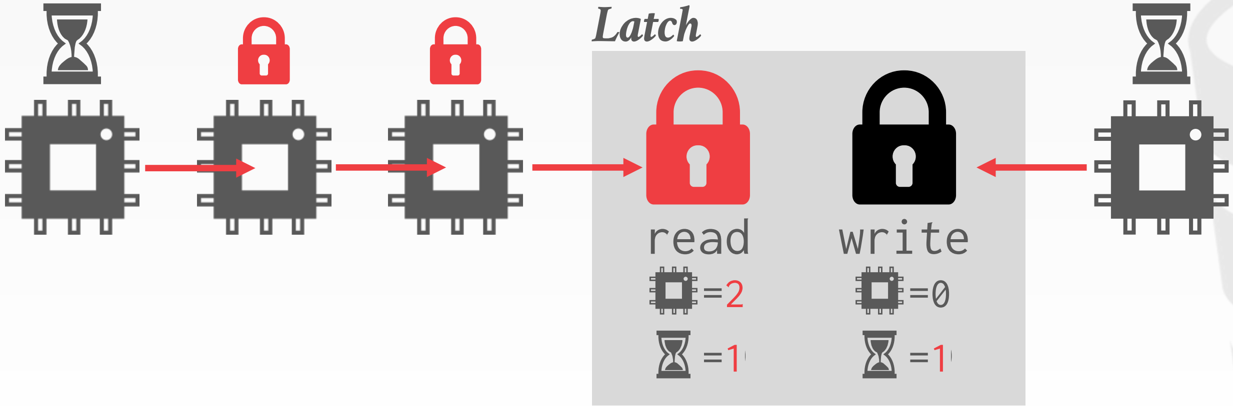
→ Allows for concurrent readers.
→ Must manage read/write queues to avoid starvation.
→ Can be implemented on top of spinlocks.



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.

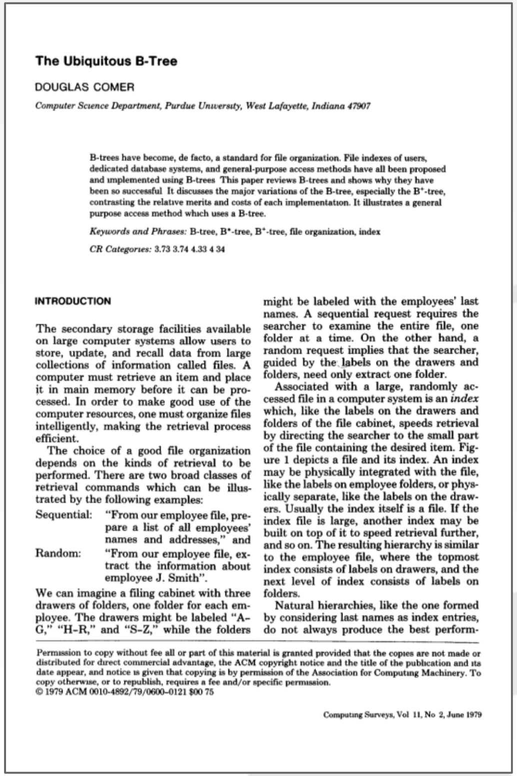


B+TREE

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.



The Ubiquitous B-Tree

DOUGLAS COMER

Computer Science Department, Purdue University, West Lafayette, Indiana 47907

B-trees have become, de facto, a standard for file organization. File indexes of users, dedicated database systems, and general-purpose access methods have all been proposed and implemented using B-trees This paper reviews B-trees and shows why they have been so successful It discusses the major variations of the B-tree, especially the B⁺-tree, contrasting the relative merits and costs of each implementation. It illustrates a general purpose access method which uses a B-tree.

Keywords and Phrases: B-tree, B*-tree, B⁺-tree, file organization, index

CR Categories: 3.73 3.74 4.33 4 34

INTRODUCTION

The secondary storage facilities available on large computer systems allow users to store, update, and recall data from large collections of information called files. A computer must retrieve an item and place it in main memory before it can be processed. In order to make good use of the computer resources, one must organize files intelligently, making the retrieval process efficient.

The choice of a good file organization depends on the kinds of retrieval to be performed. There are two broad classes of retrieval commands which can be illustrated by the following examples:

Sequential: "From our employee file, prepare a list of all employees' names and addresses," and

Random: "From our employee file, extract the information about employee J. Smith".

We can imagine a filing cabinet with three drawers of folders, one folder for each employee. The drawers might be labeled "A–G," "H–R," and "S–Z," while the folders might be labeled with the employees' last names. A sequential request requires the searcher to examine the entire file, one folder at a time. On the other hand, a random request implies that the searcher, guided by the labels on the drawers and folders, need only extract one folder.

Associated with a large, randomly accessed file in a computer system is an *index* which, like the labels on the drawers and folders of the file cabinet, speeds retrieval by directing the searcher to the small part of the file containing the desired item. Figure 1 depicts a file and its index. An index may be physically integrated with the file, like the labels on employee folders, or physically separate, like the labels on the drawers. Usually the index itself is a file. If the index file is large, another index may be built on top of it to speed retrieval further, and so on. The resulting hierarchy is similar to the employee file, where the topmost index consists of labels on drawers, and the next level of index consists of labels on folders.

Natural hierarchies, like the one formed by considering last names as index entries, do not always produce the best perform-

LATCH CRABBING /COUPLING

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

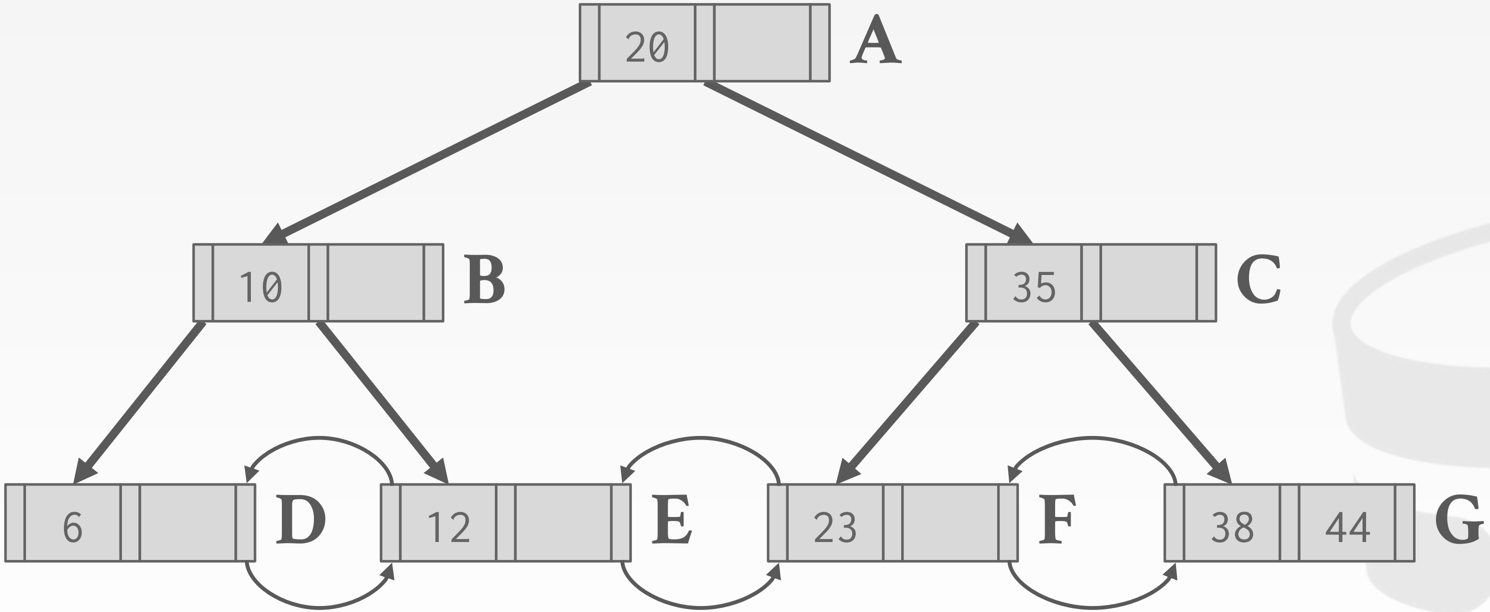
A thread can release latch on a parent node if its child node considered **safe**.
→ Any node that won't split or merge when updated.
→ Not full (on insertion)
→ More than half-full (on deletion)

LATCH CRABBING

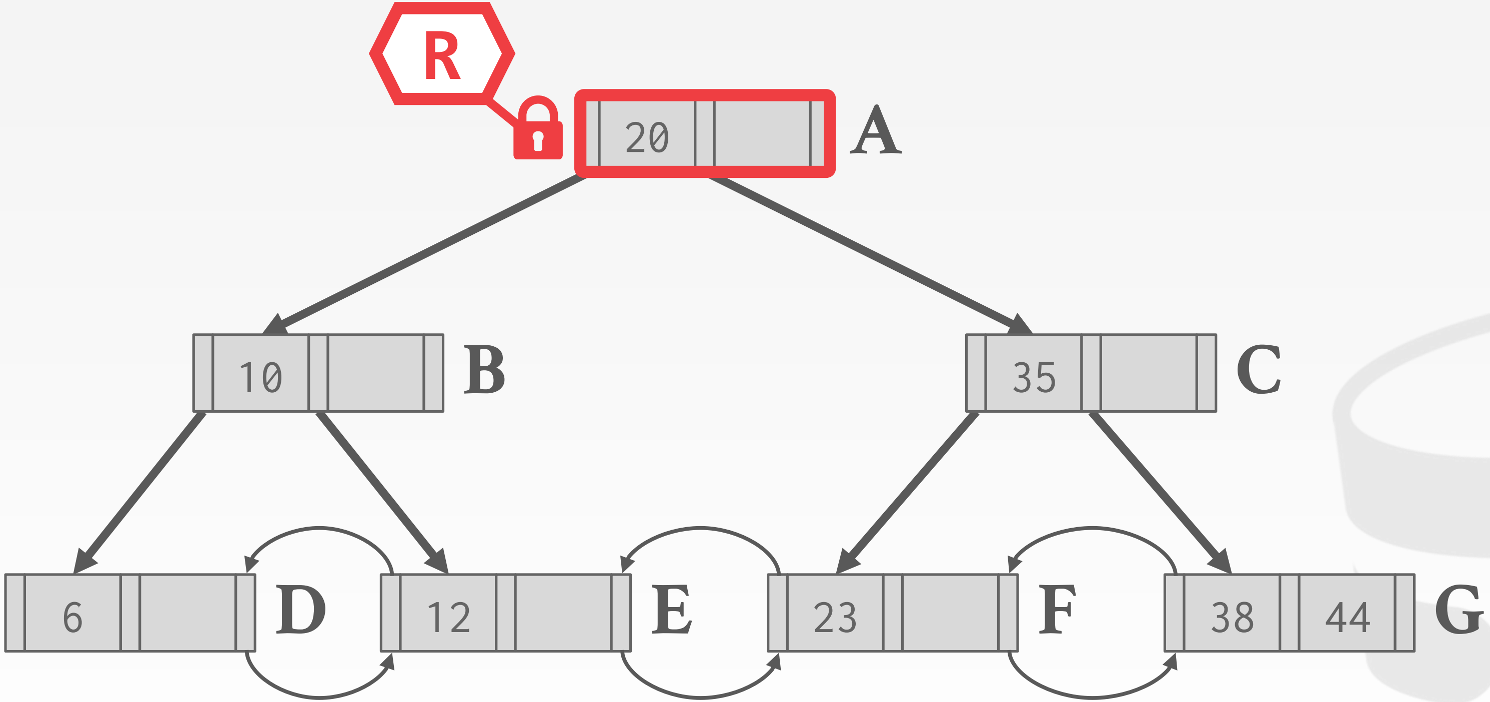
Search: Start at root and go down; repeatedly,
→ Acquire read (**R**) latch on child
→ Then unlock the parent node.

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

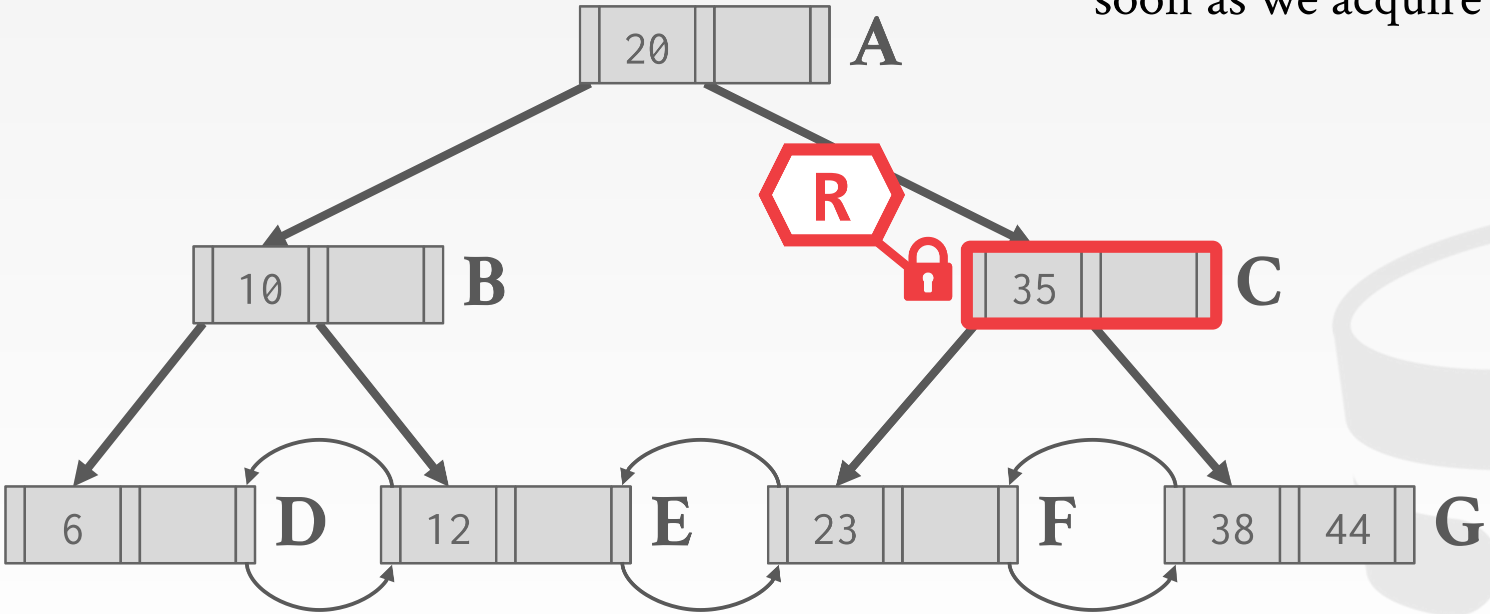


EXAMPLE #1: SEARCH 23



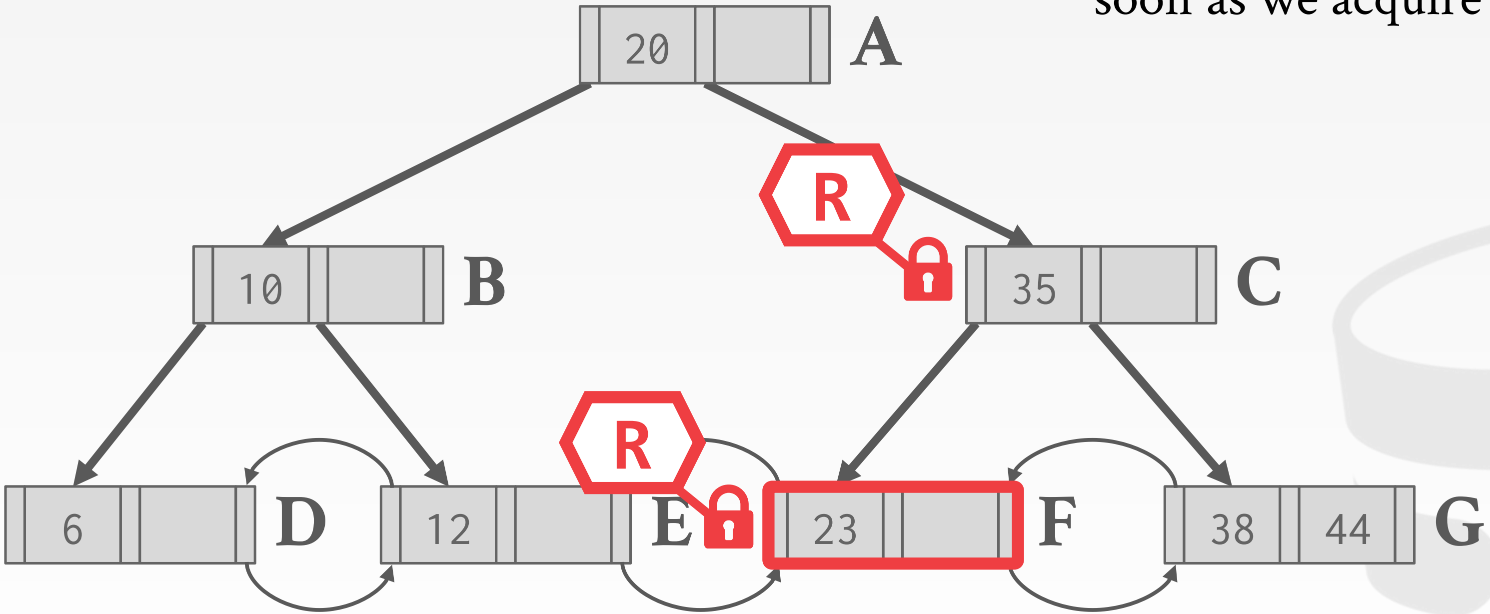
EXAMPLE #1: SEARCH 23

We can release the latch on A as soon as we acquire the latch for C.



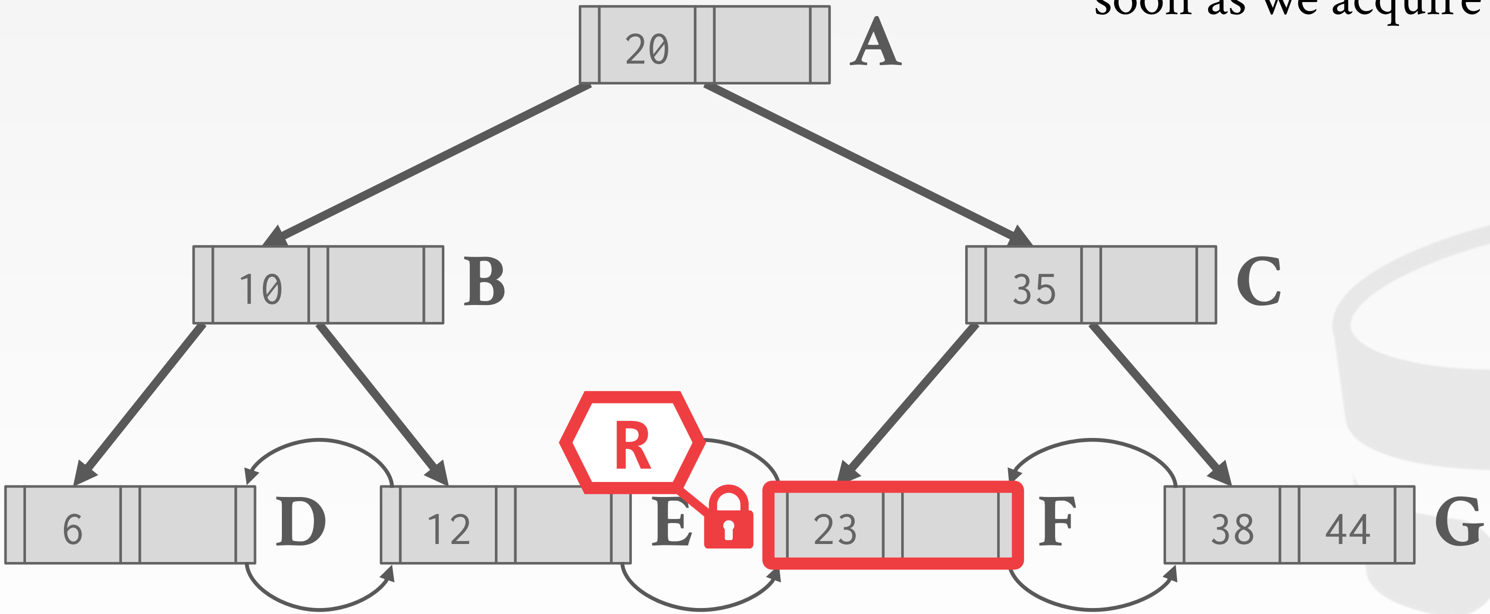
EXAMPLE #1: SEARCH 23

We can release the latch on **A** as soon as we acquire the latch for **C**.

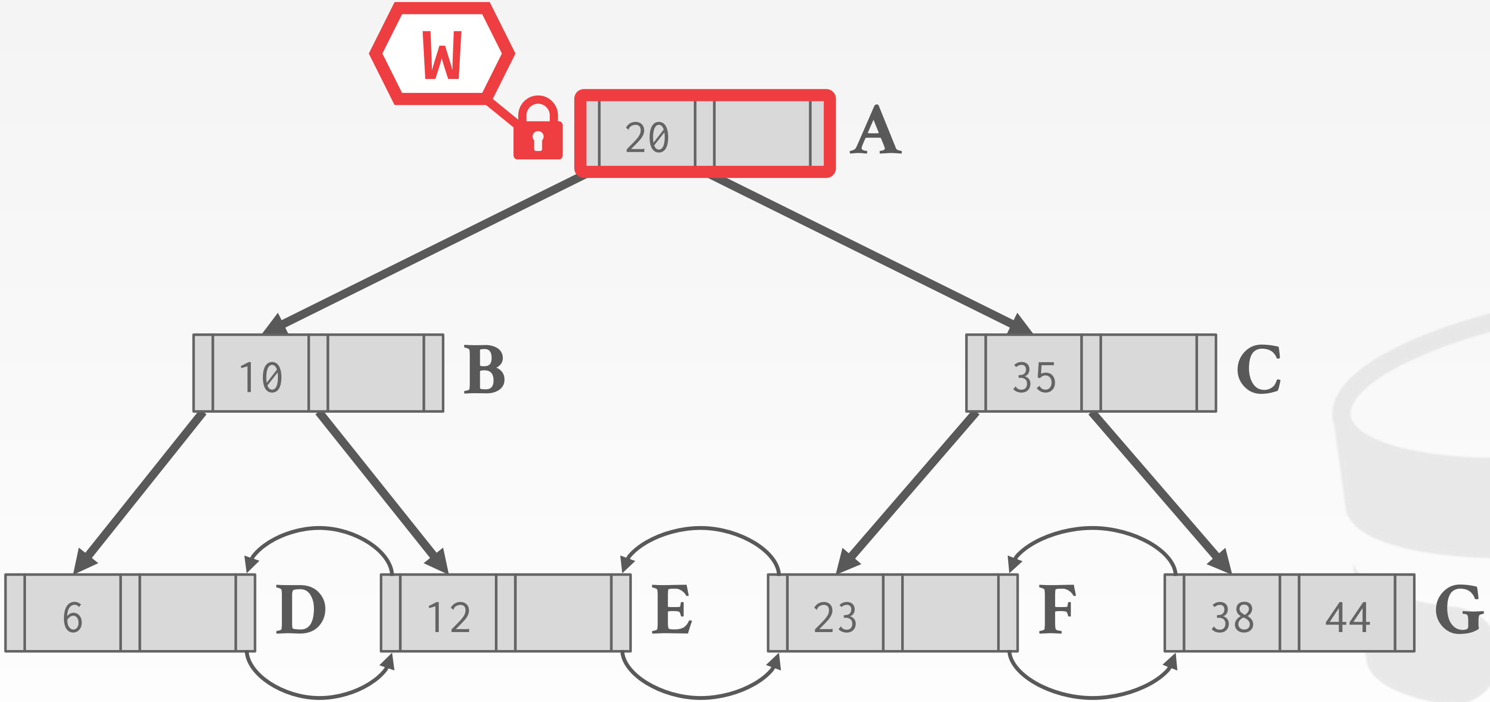


EXAMPLE #1: SEARCH 23

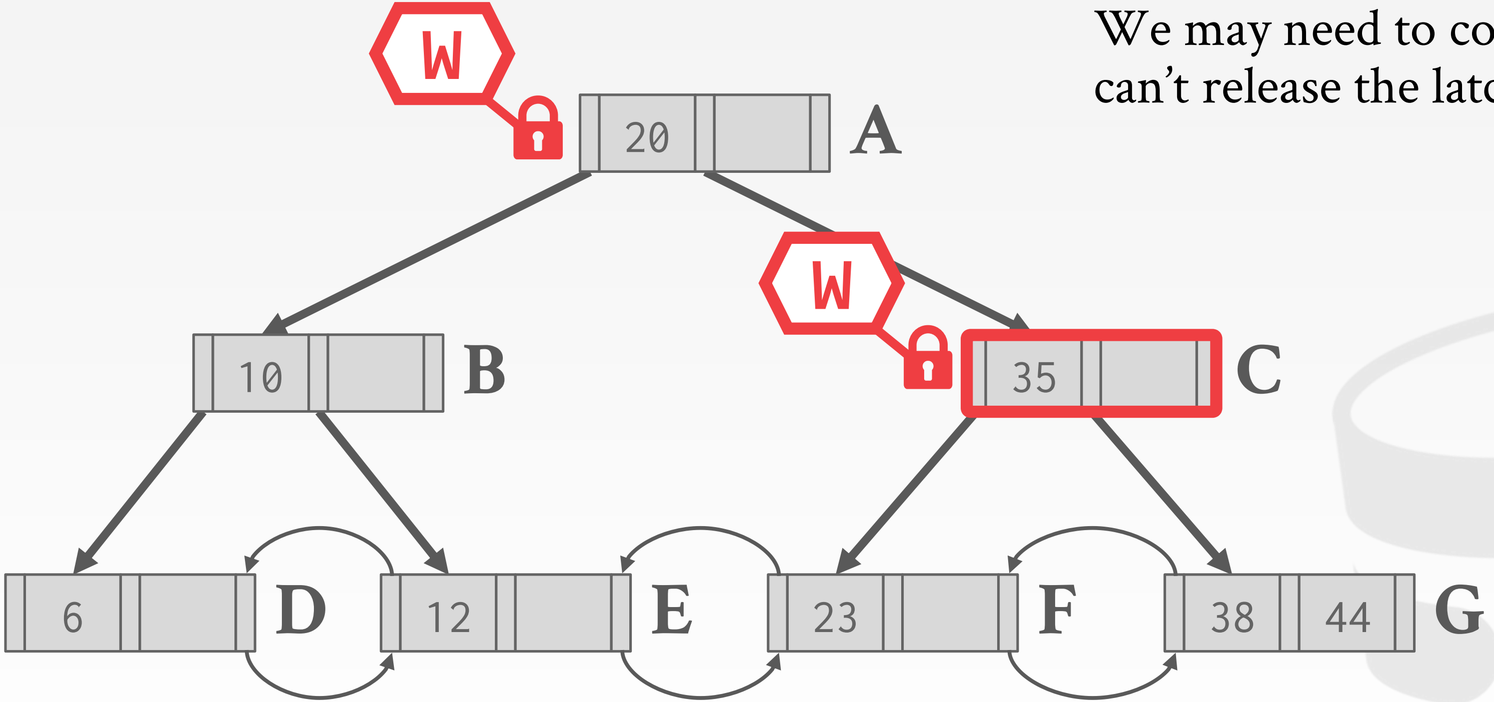
We can release the latch on **A** as soon as we acquire the latch for **C**.



EXAMPLE #2: DELETE 44

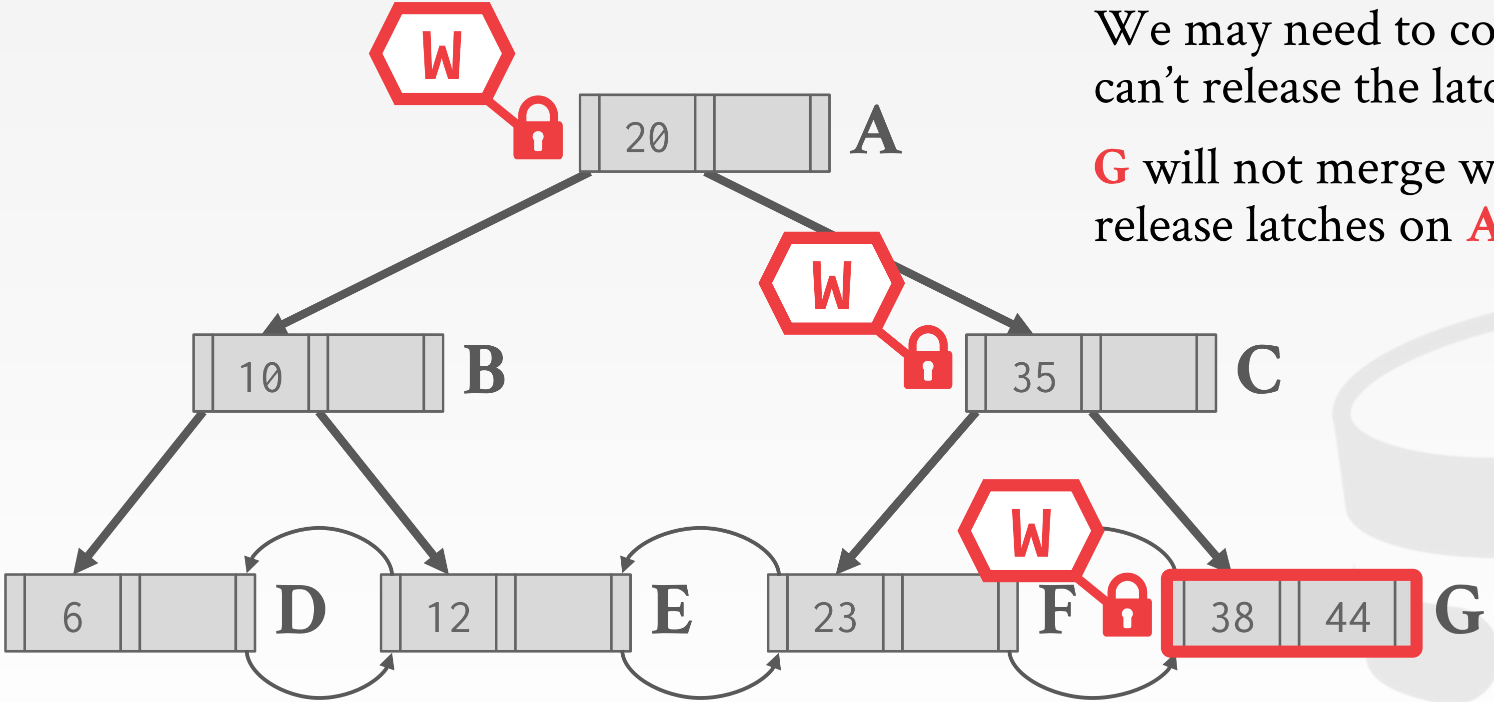


EXAMPLE #2: DELETE 44



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

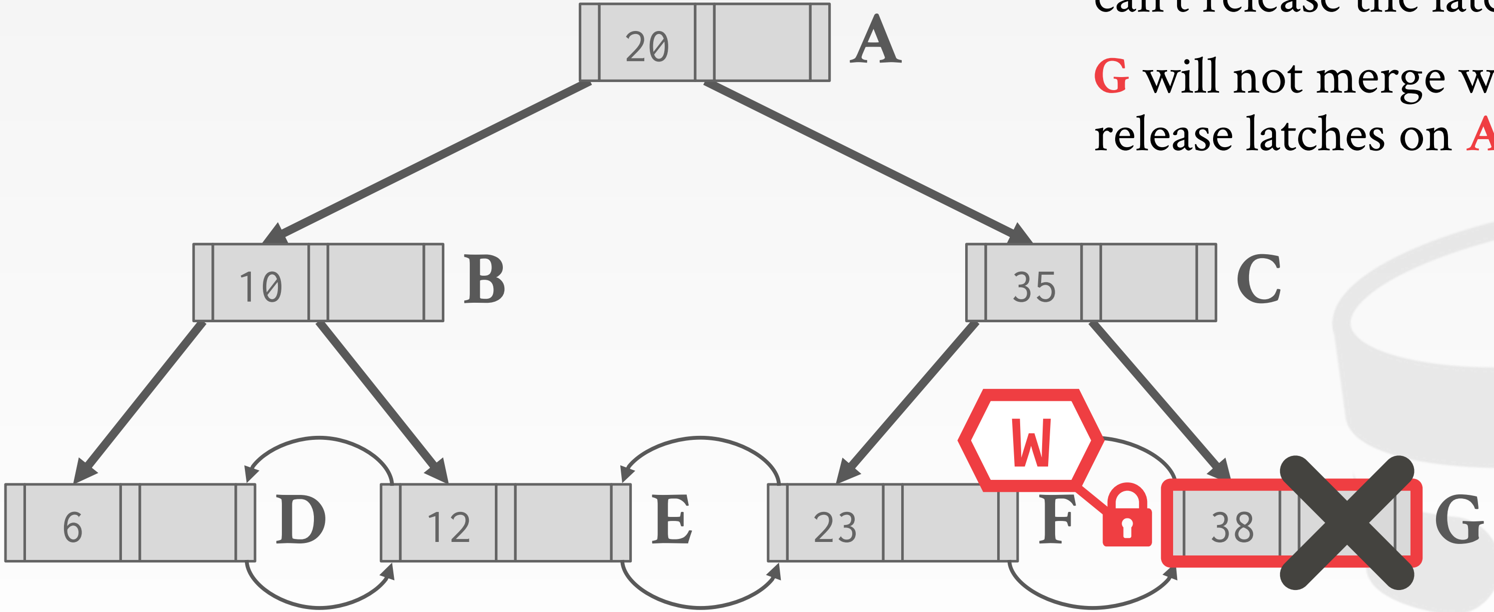
EXAMPLE #2: DELETE 44



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.

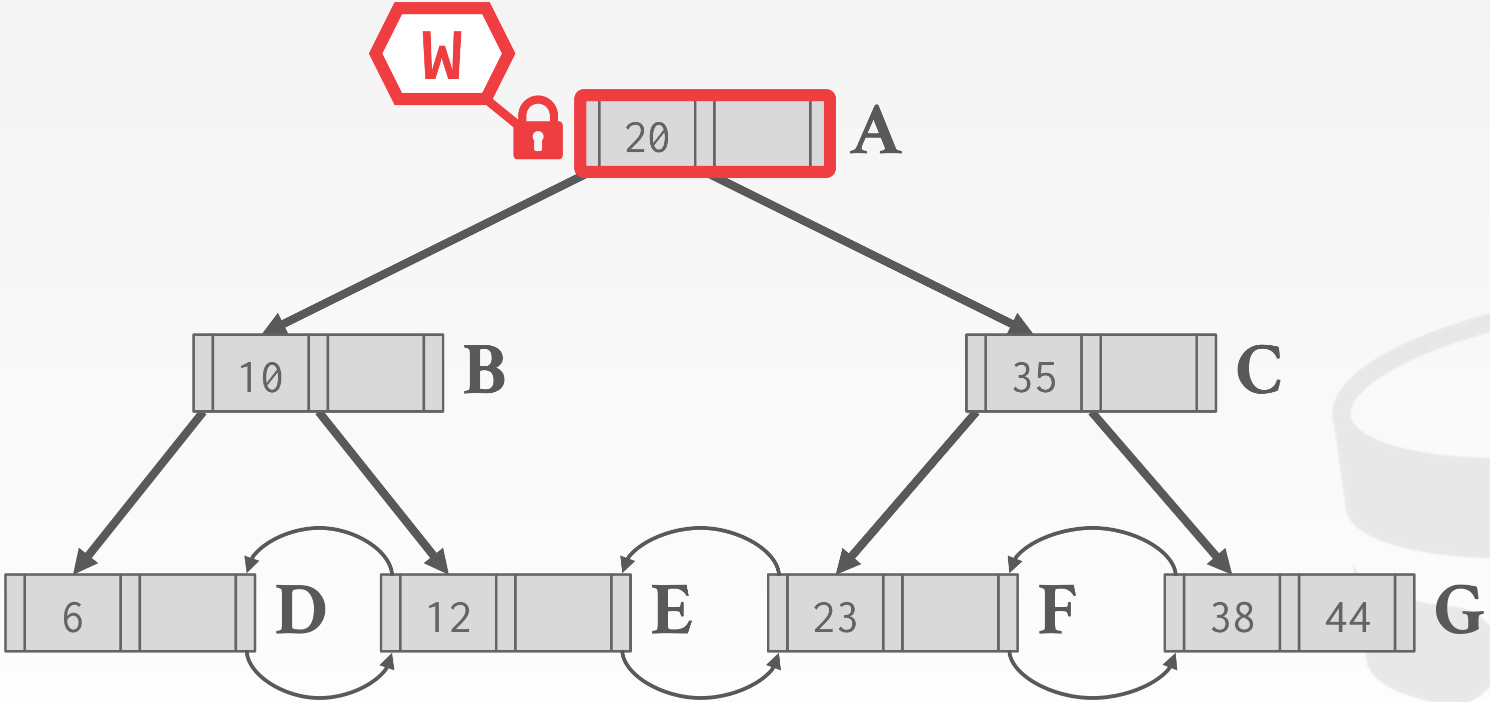
EXAMPLE #2: DELETE 44



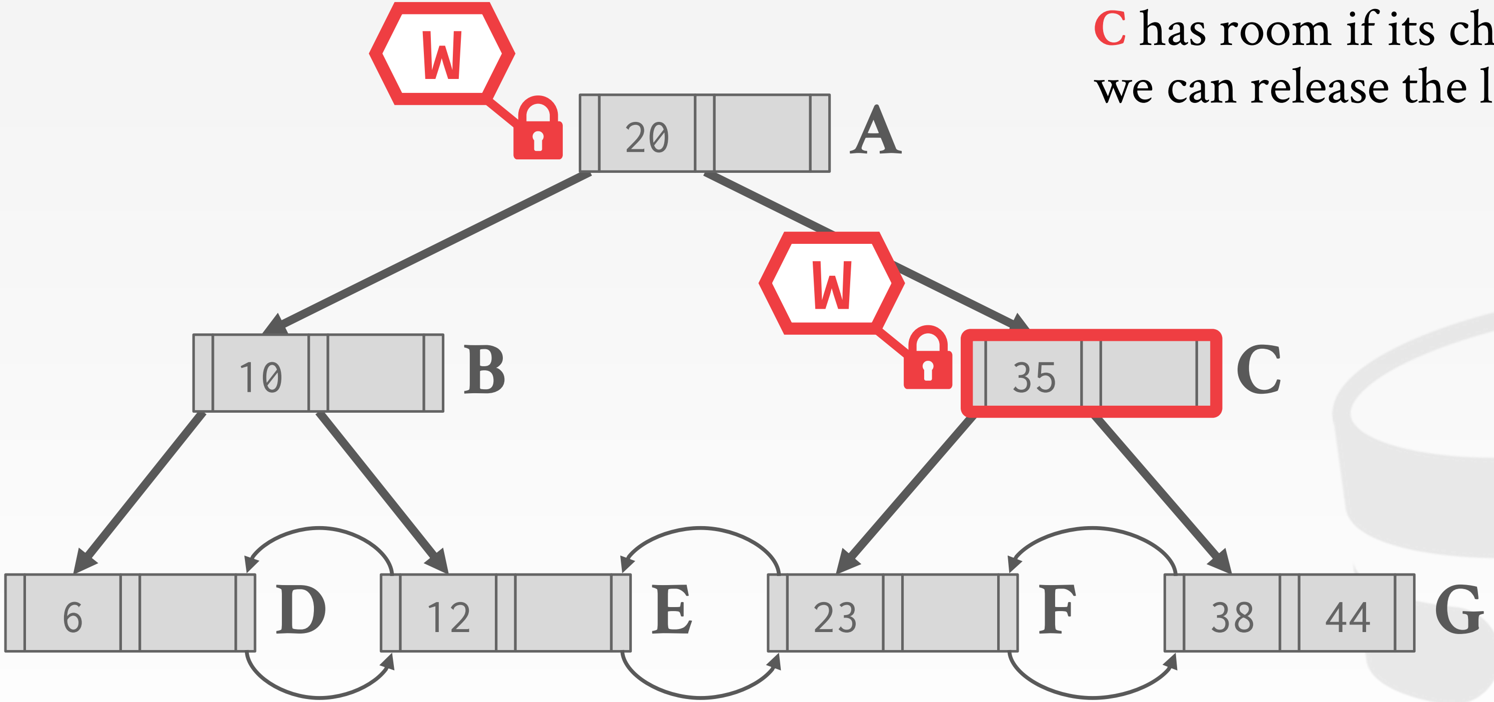
We may need to coalesce **C**, so we can't release the latch on **A**.

G will not merge with **F**, so we can release latches on **A** and **C**.

EXAMPLE #3: INSERT 40



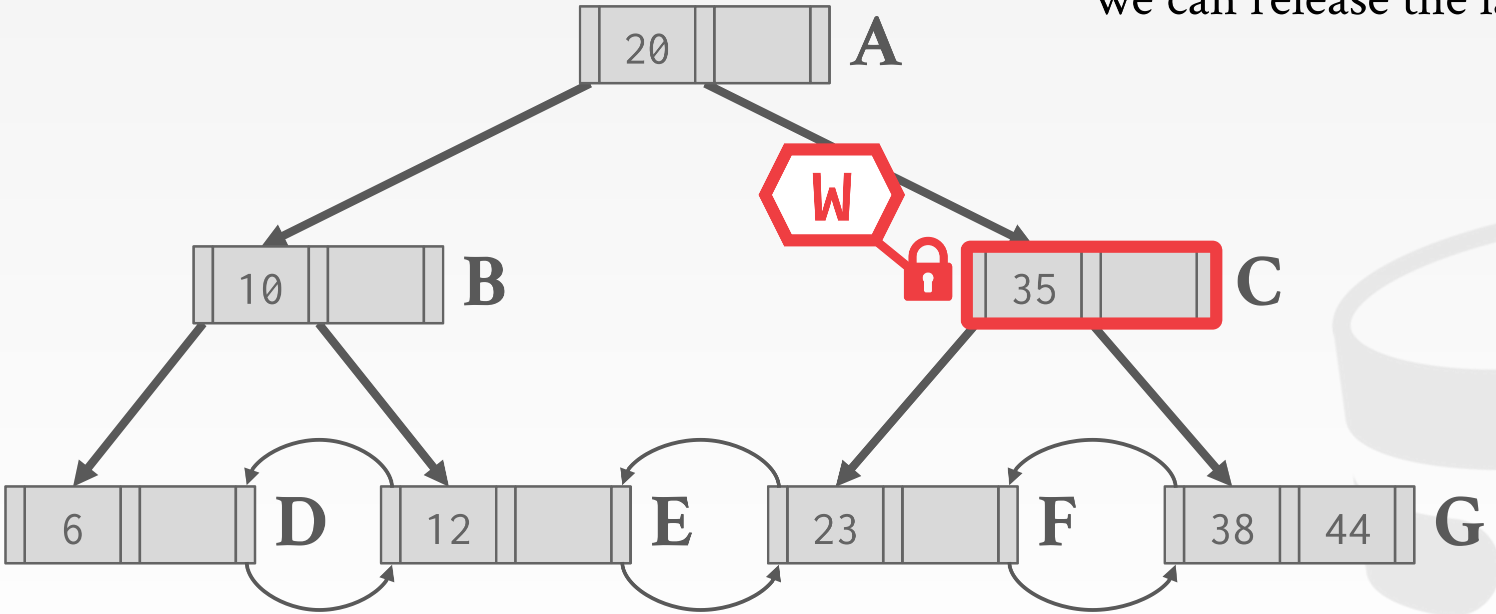
EXAMPLE #3: INSERT 40



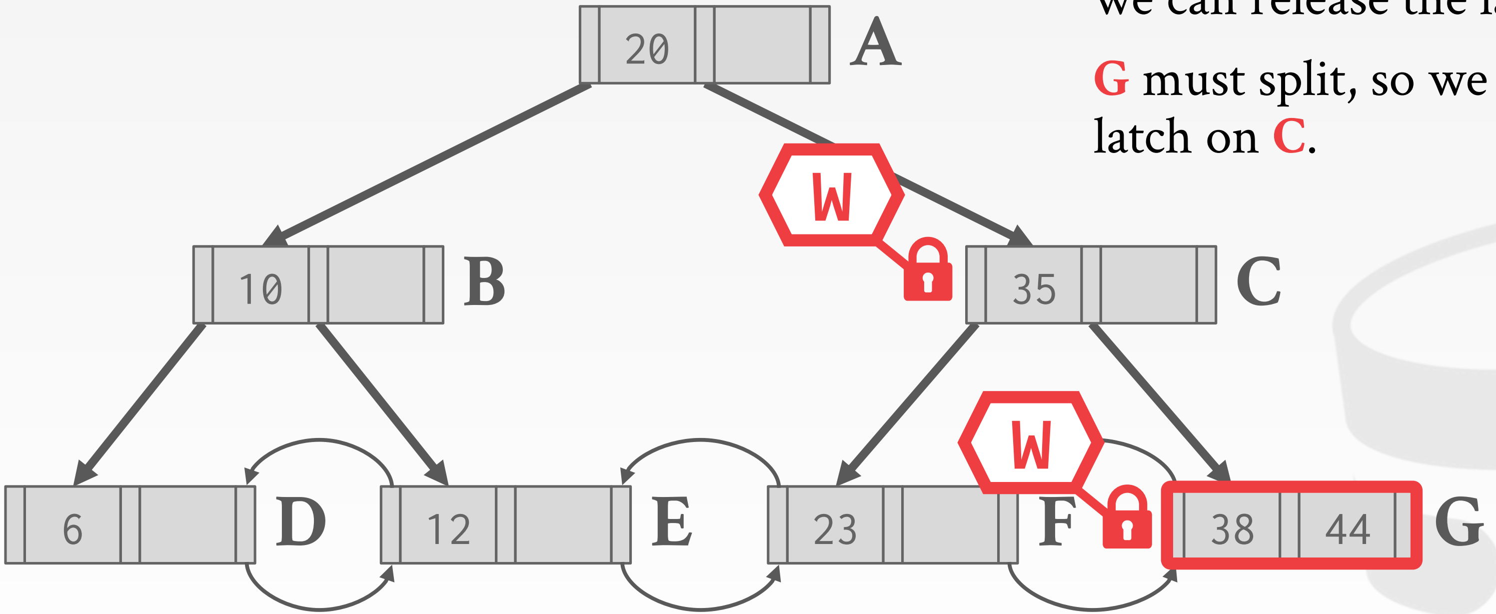
C has room if its child has to split, so we can release the latch on A.

EXAMPLE #3: INSERT 40

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



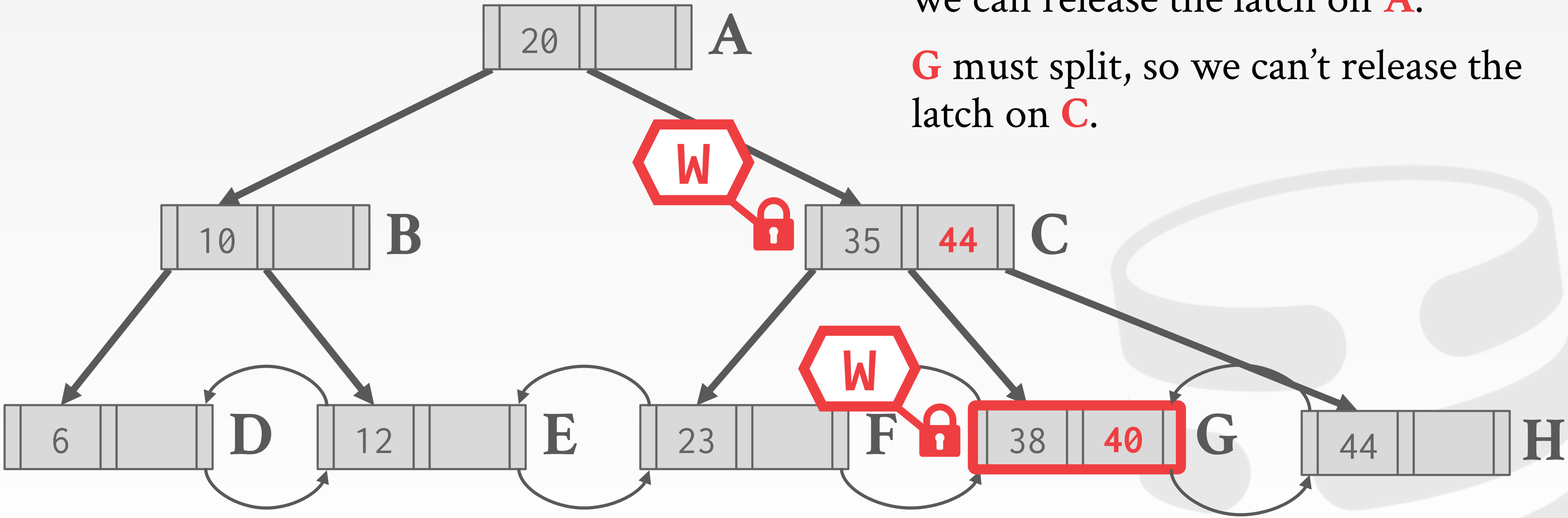
EXAMPLE #3: INSERT 40



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.

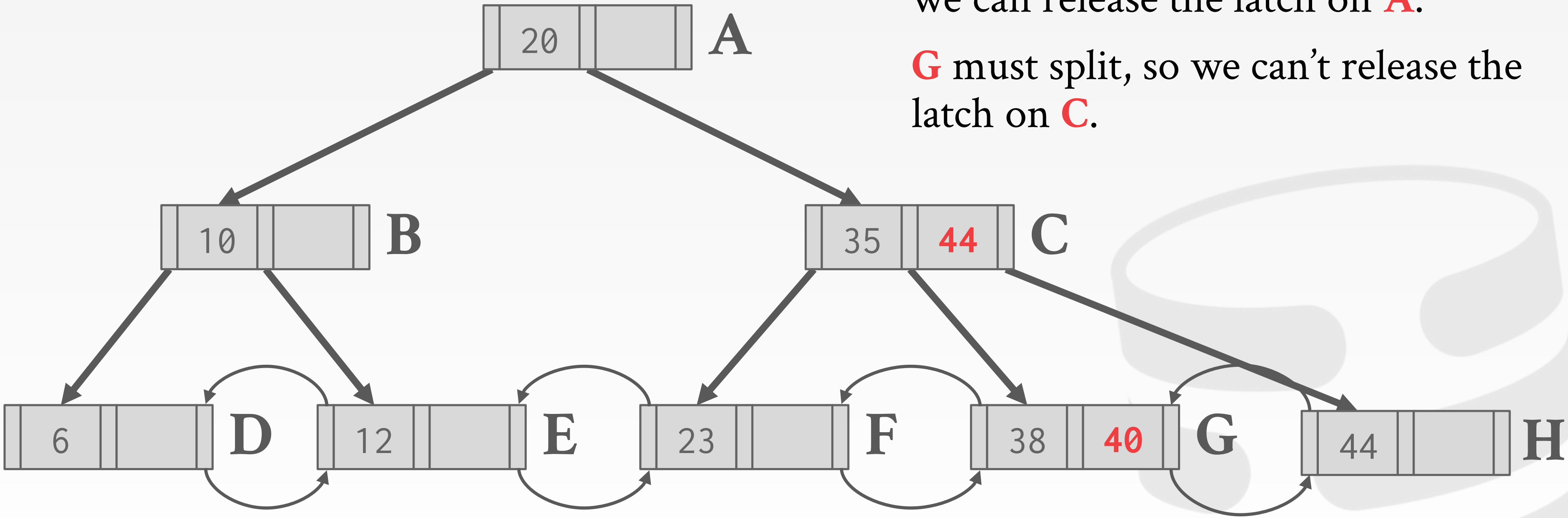
EXAMPLE #3: INSERT 40



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**.

BETTER LATCH CRABBING

The basic latch crabbing algorithm always takes a write latch on the root for any update.

→ This makes the index essentially single threaded.

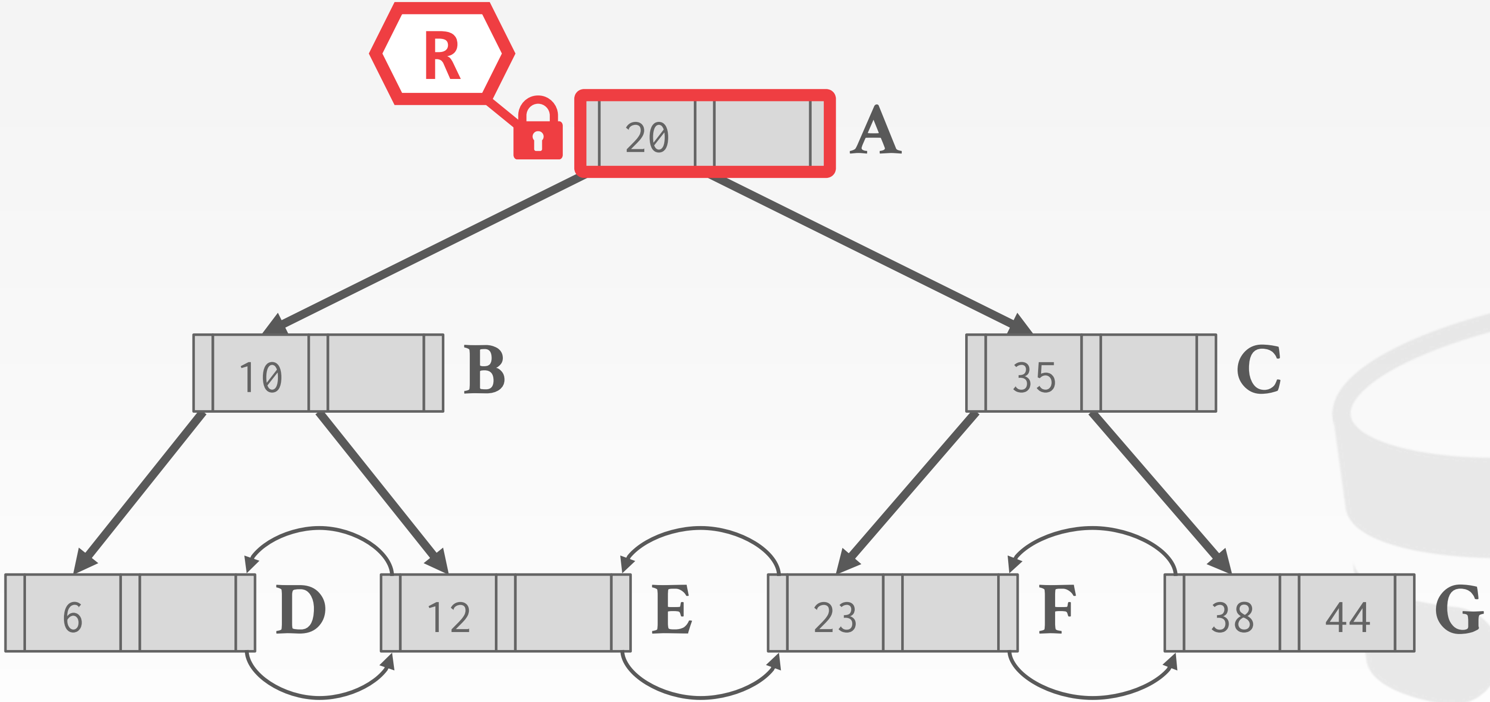
A better approach is to optimistically assume that the target leaf node is safe.

→ Take **R** latches as you traverse the tree to reach it and verify.

→ If leaf is not safe, then do previous algorithm.

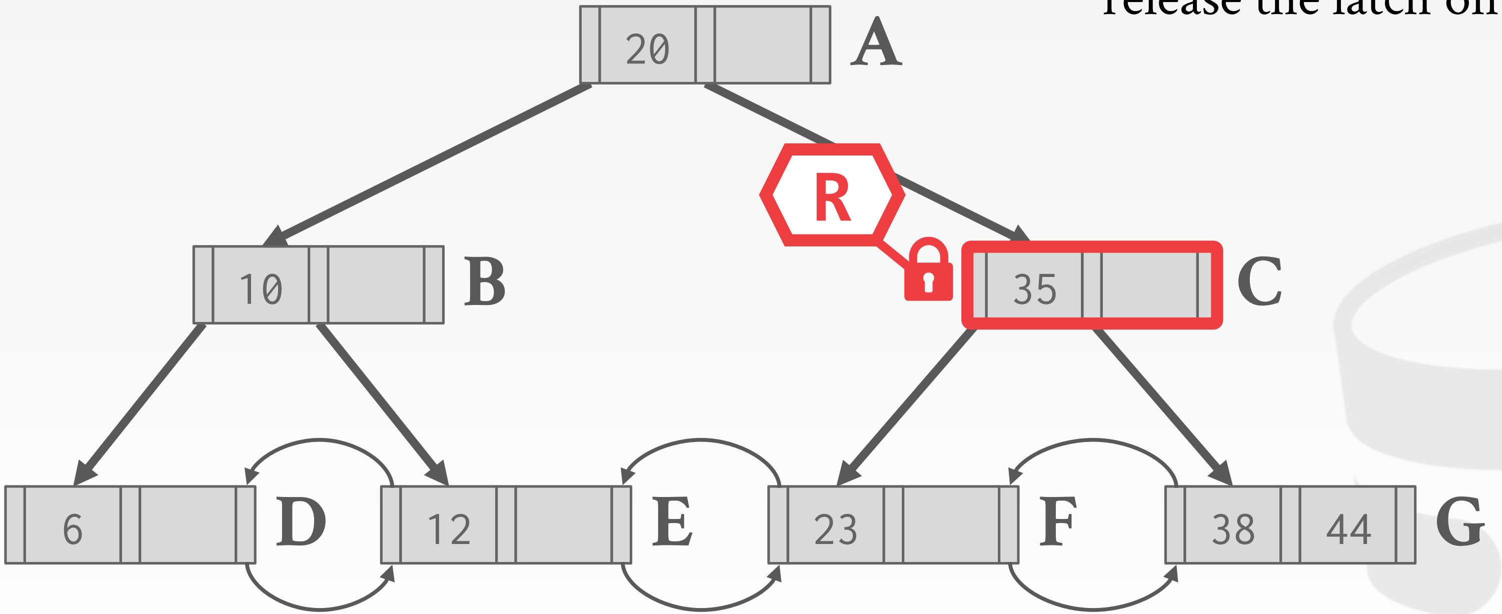
CONCURRENCY OF OPERATIONS ON B-TREES
ACTA INFORMATICA 1977

EXAMPLE #4: DELETE 44



EXAMPLE #4: DELETE 44

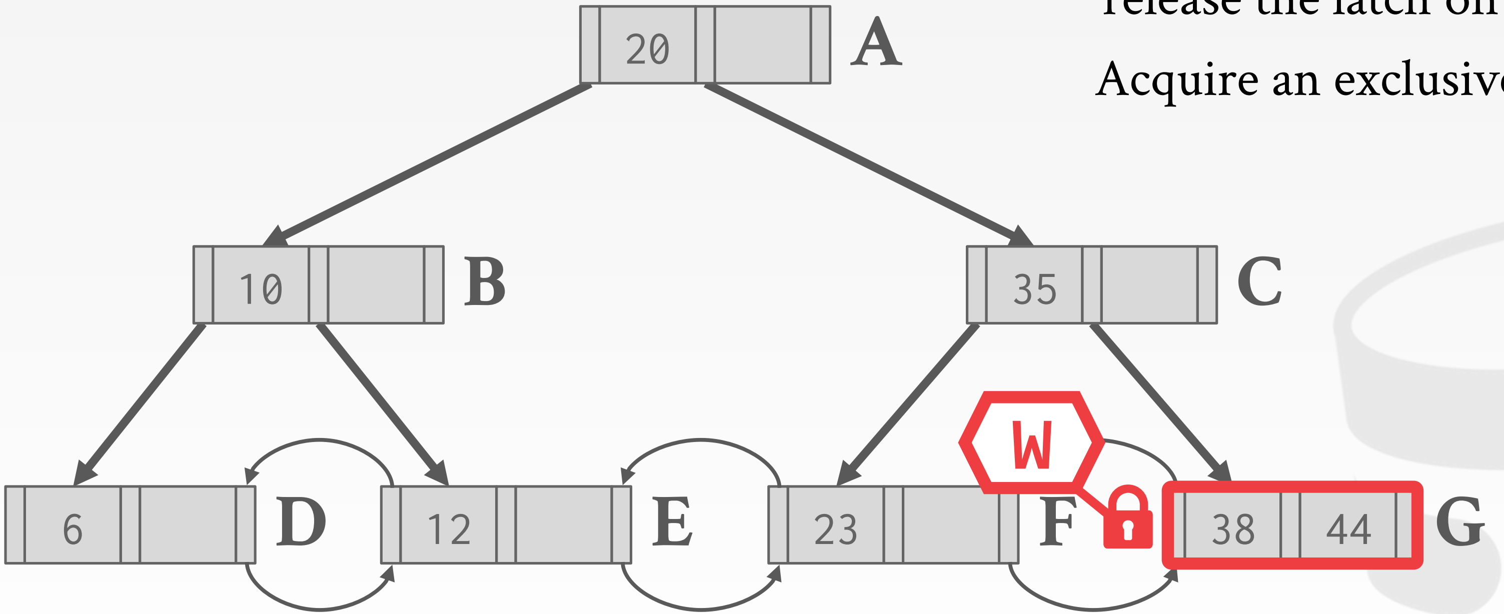
We assume that C is safe, so we can release the latch on A.



EXAMPLE #4: DELETE 44

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

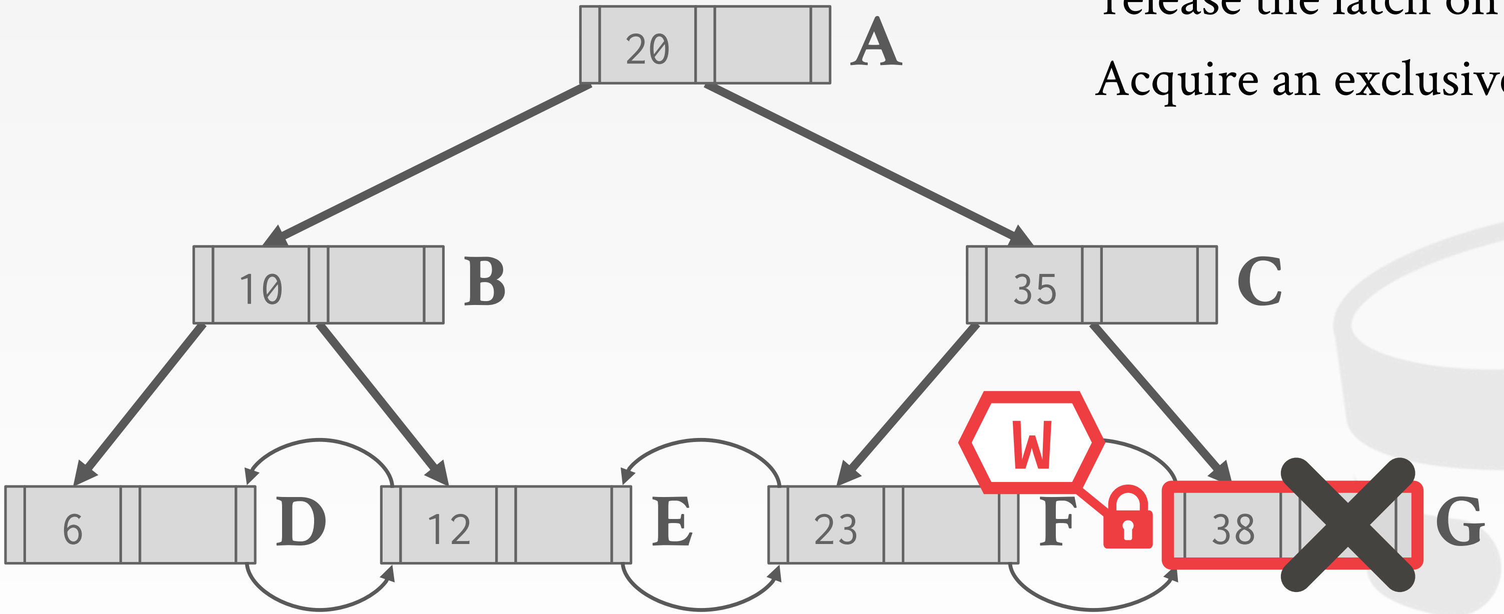
Acquire an exclusive latch on G.



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

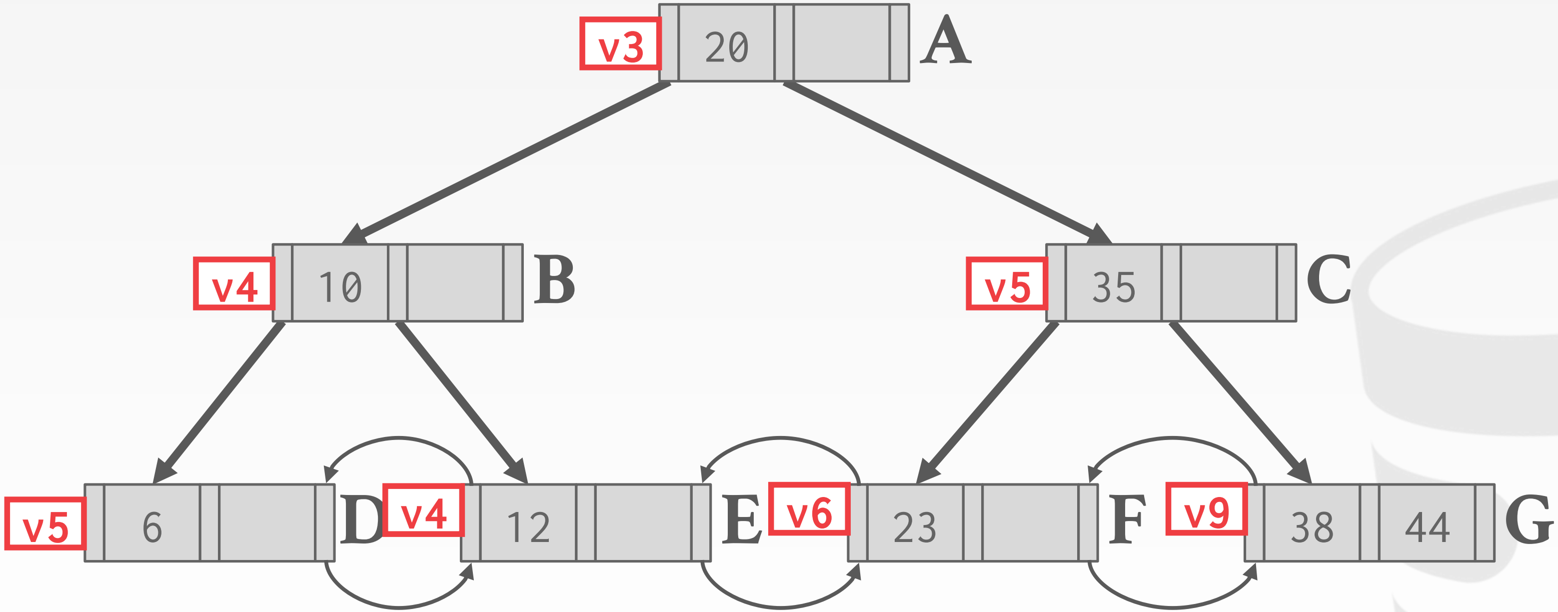
Optimistic crabbing scheme where writers are not blocked on readers.

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.

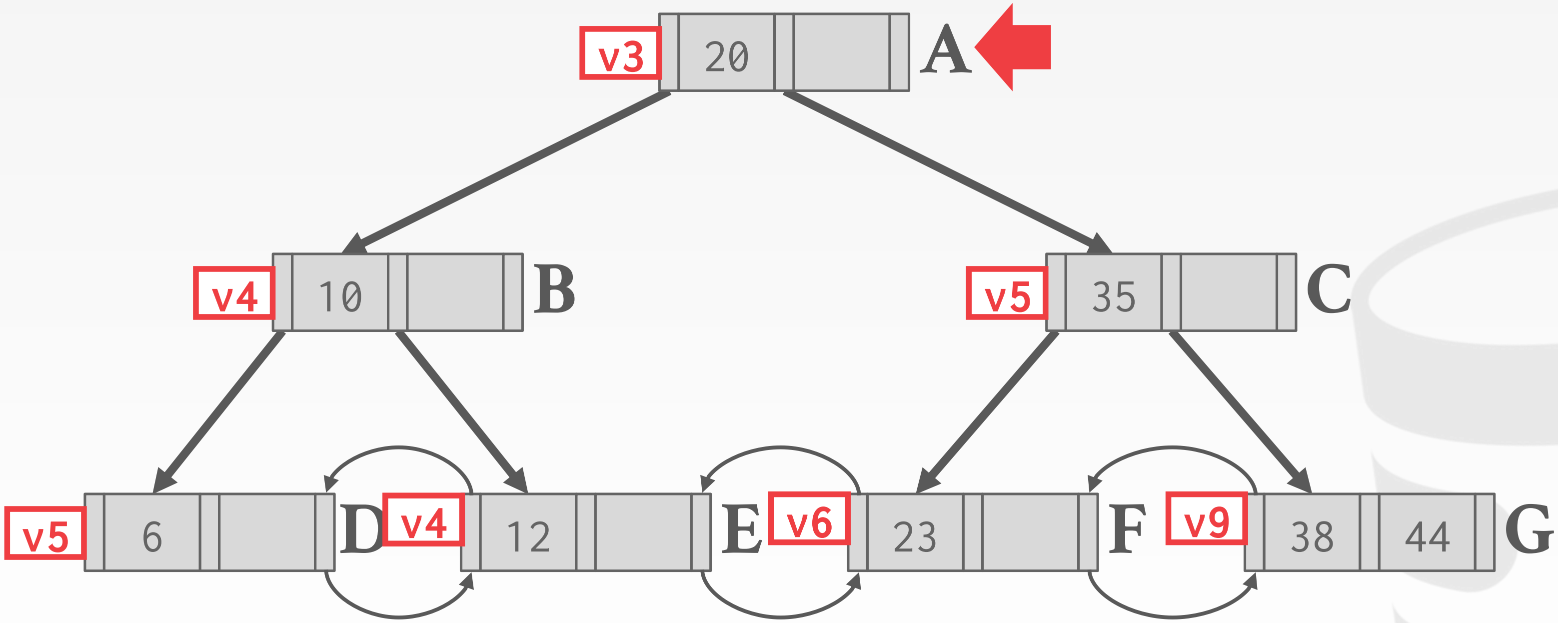
THE ART OF PRACTICAL SYNCHRONIZATION
DAMON 2016

VERSIONED LATCHES: SEARCH 44

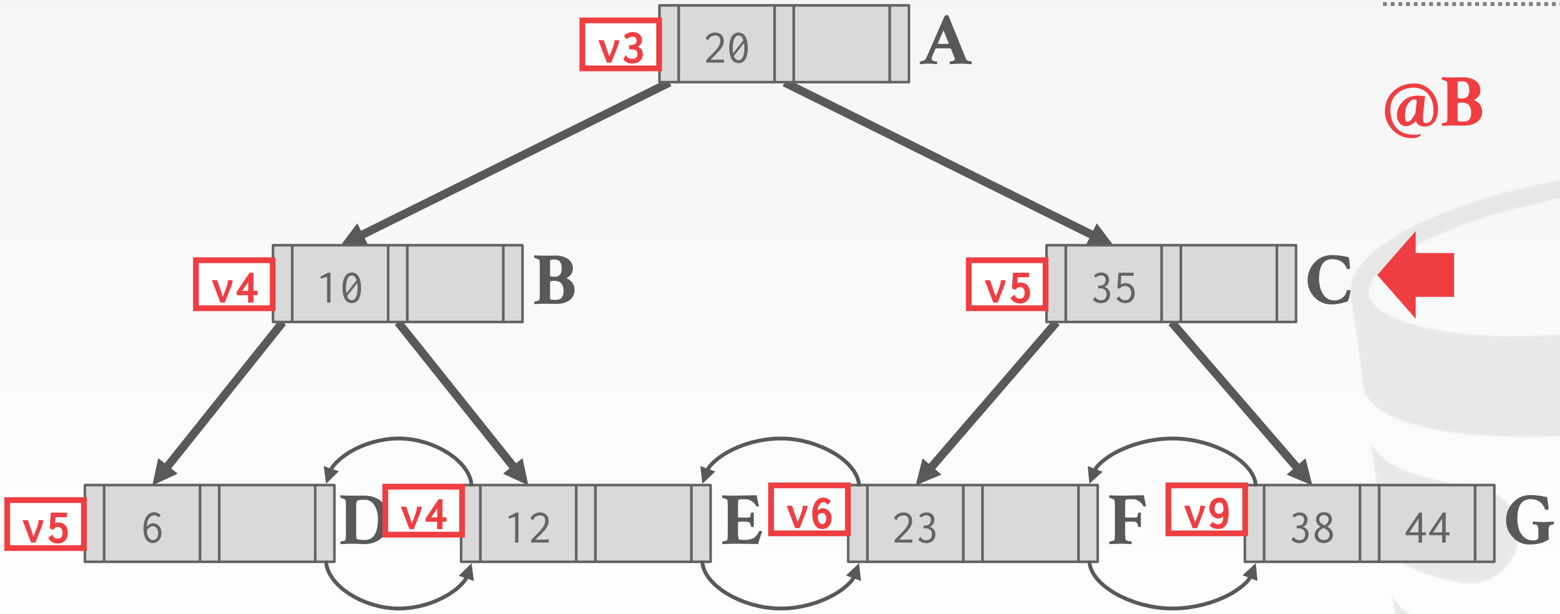


VERSIONED LATCHES: SEARCH 44

@A A: Read v3



VERSIONED LATCHES: SEARCH 44



@A A: Read v3
A: Examine Node

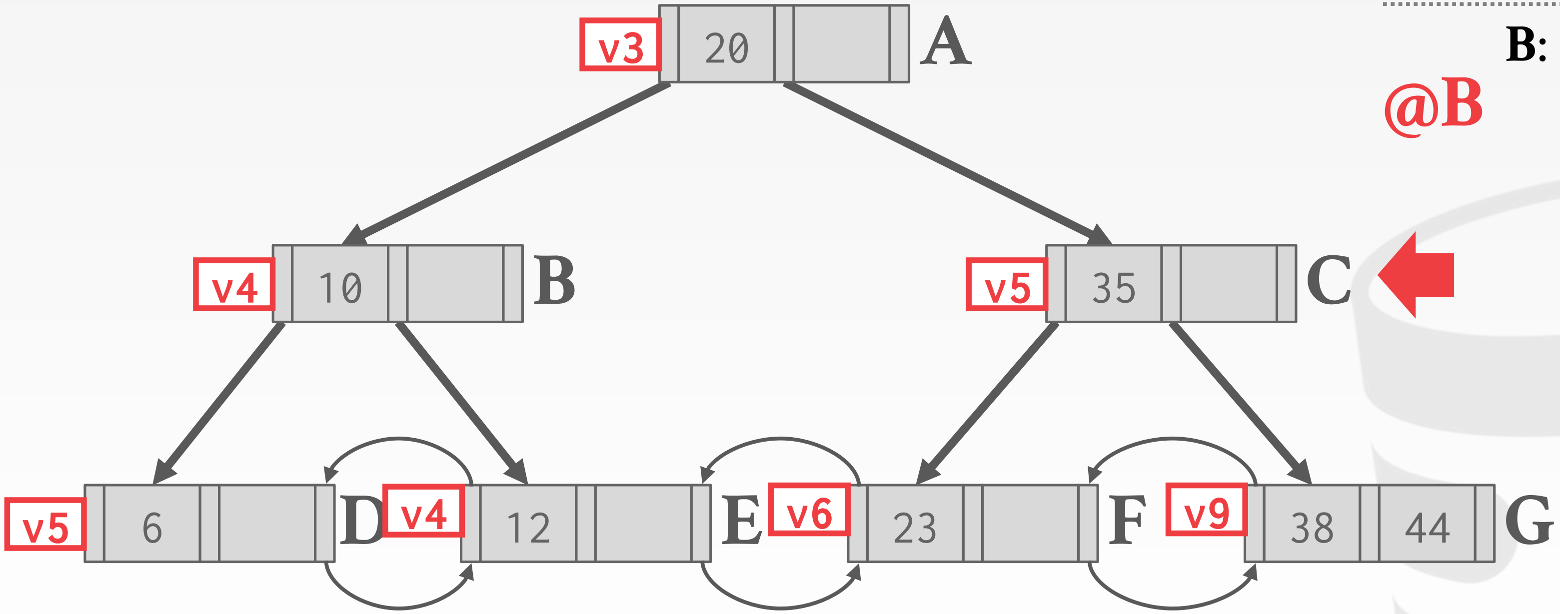
@B

VERSIONED LATCHES: SEARCH 44

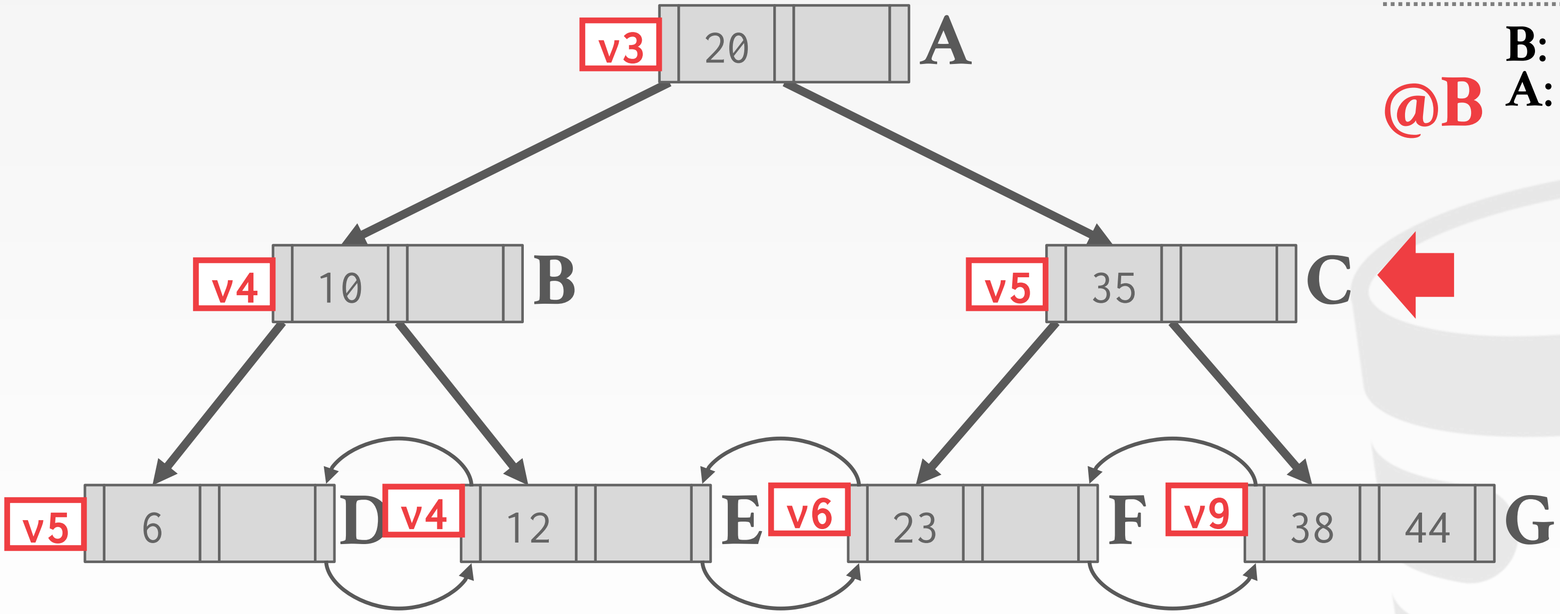
@A
A: Read v3
A: Examine Node

B: Read v5

@B



VERSIONED LATCHES: SEARCH 44



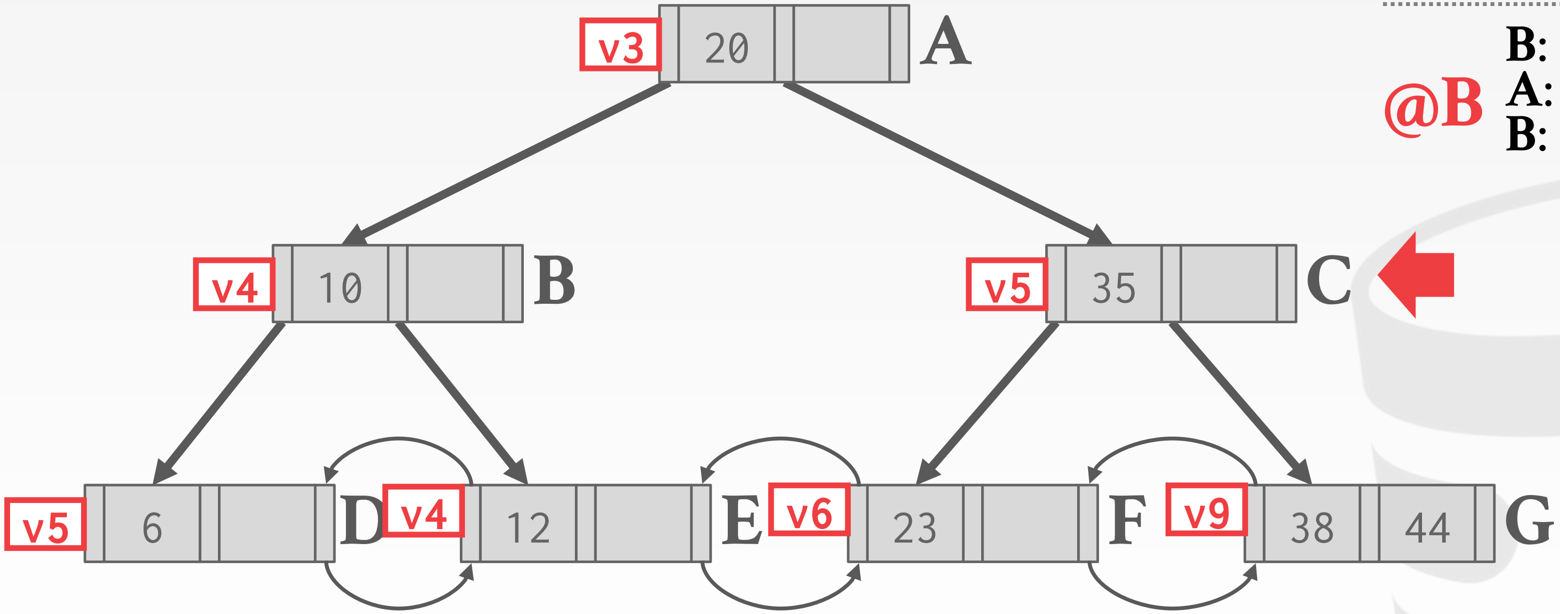
@A
A: Read v3
A: Examine Node

@B
B: Read v5
A: Recheck v3

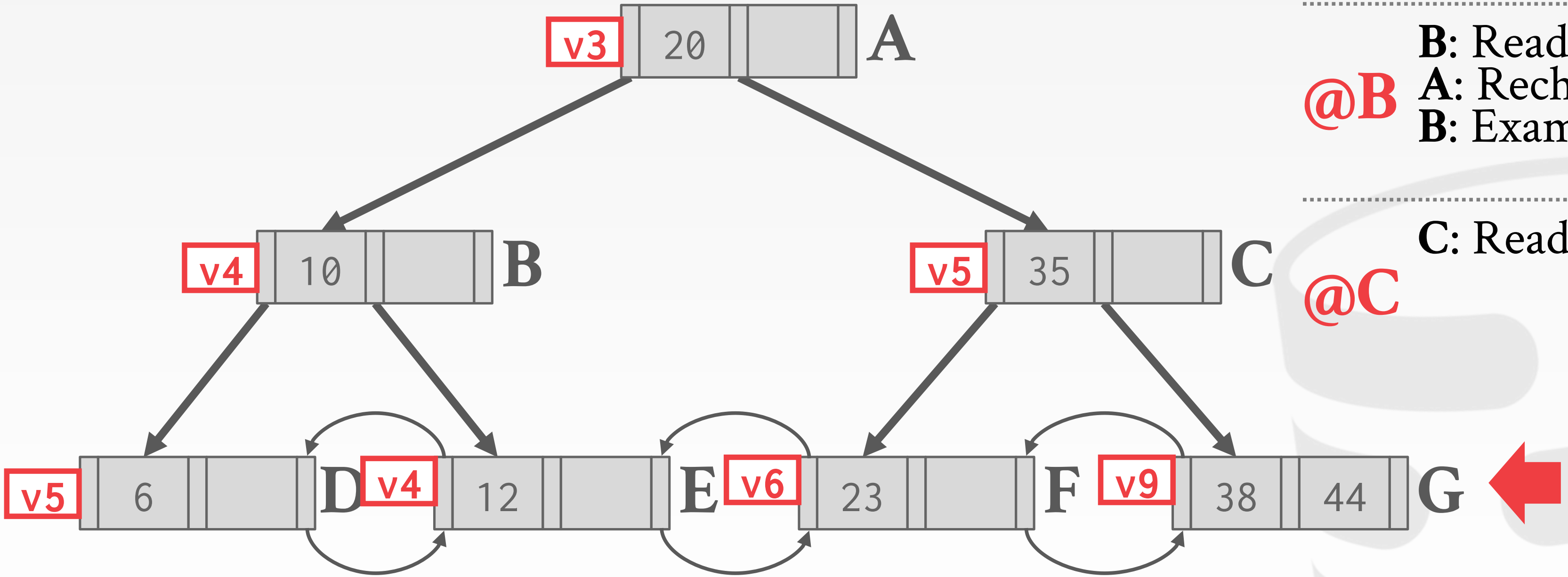
VERSIONED LATCHES: SEARCH 44

@A
A: Read v3
A: Examine Node

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



VERSIONED LATCHES: SEARCH 44

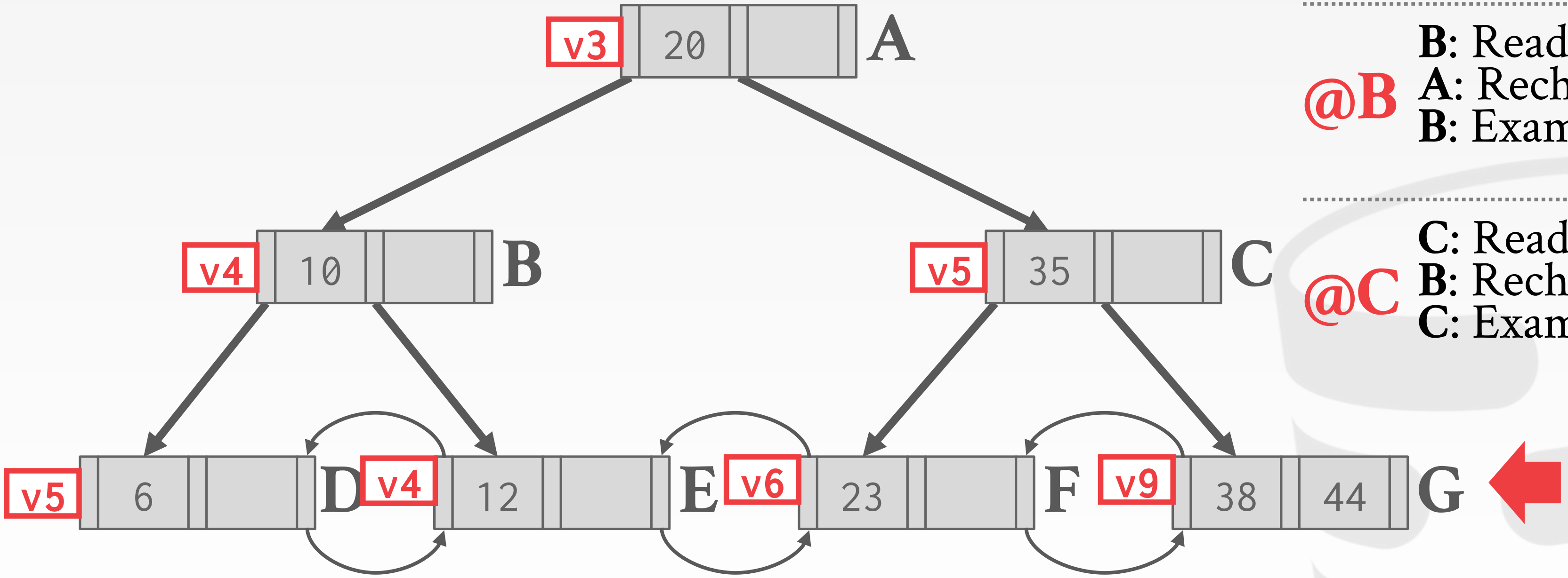


@A
A: Read v3
A: Examine Node

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

@C
C: Read v9

VERSIONED LATCHES: SEARCH 44

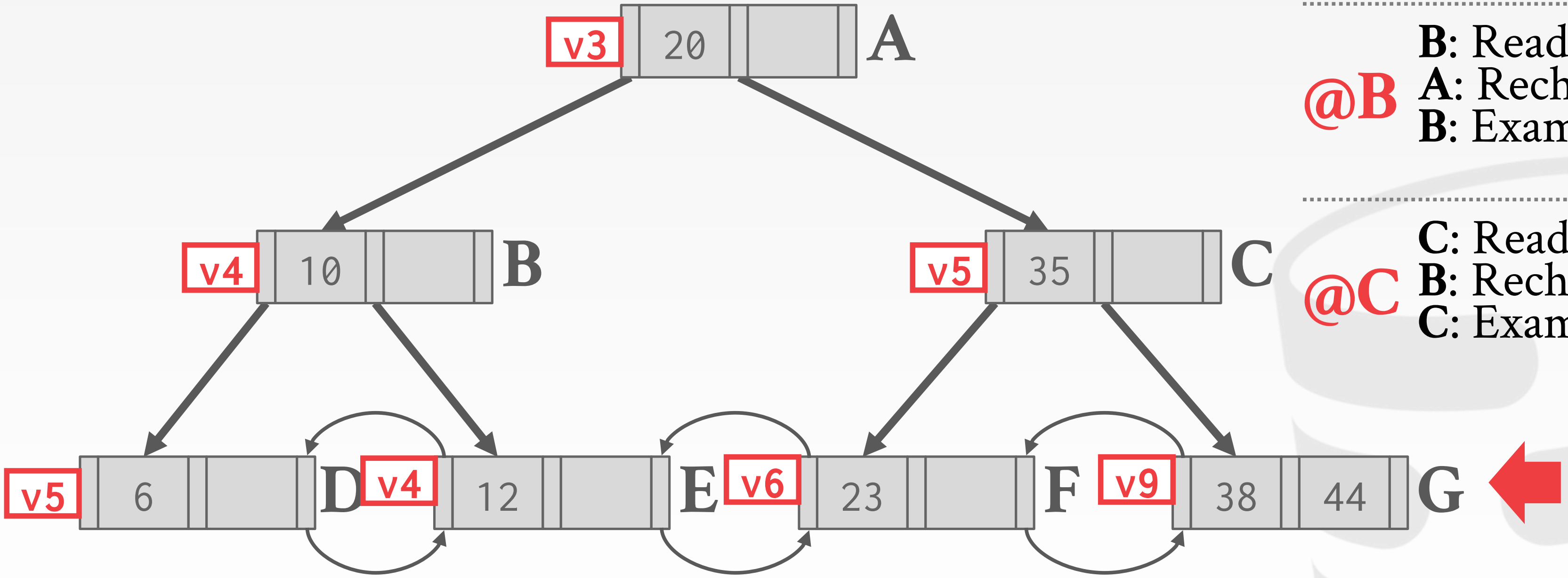


@A
A: Read v3
A: Examine Node

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

@C
C: Read v9
B: Recheck v5
C: Examine Node

VERSIONED LATCHES: SEARCH 44

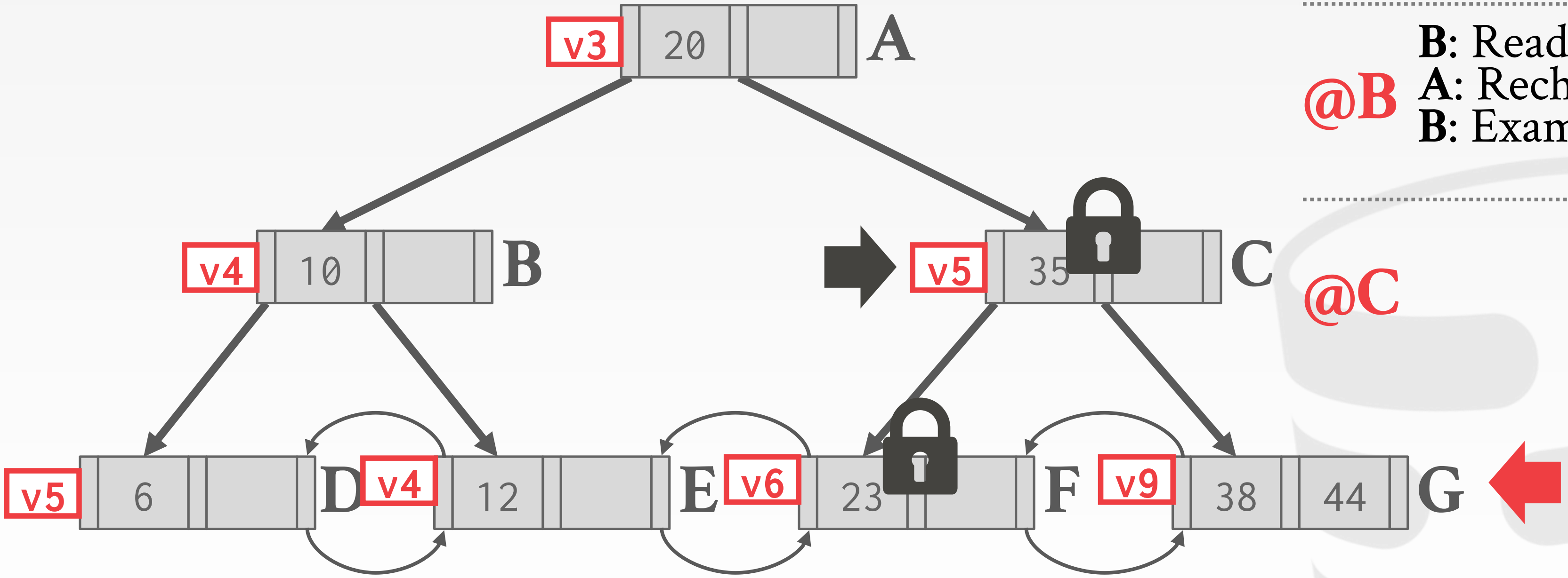


@A
A: Read v3
A: Examine Node

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

@C
C: Read v9
B: Recheck v5
C: Examine Node

VERSIONED LATCHES: SEARCH 44

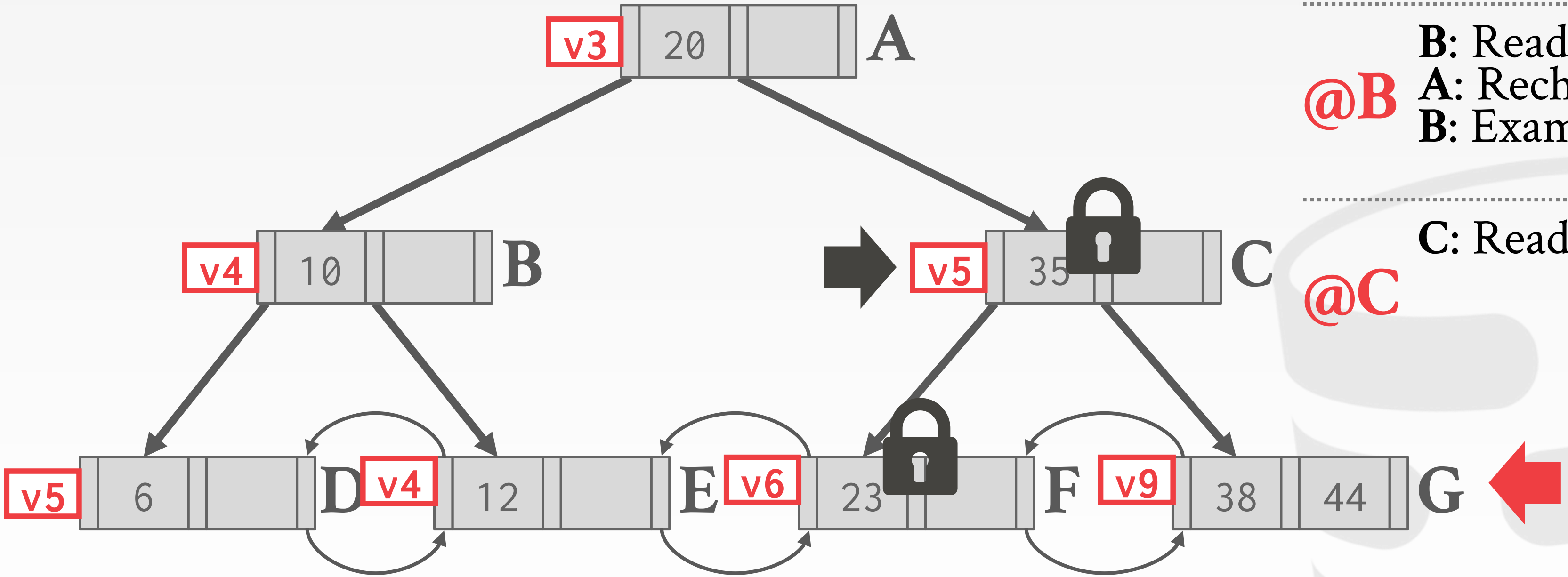


@A
A: Read v3
A: Examine Node

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

@C

VERSIONED LATCHES: SEARCH 44

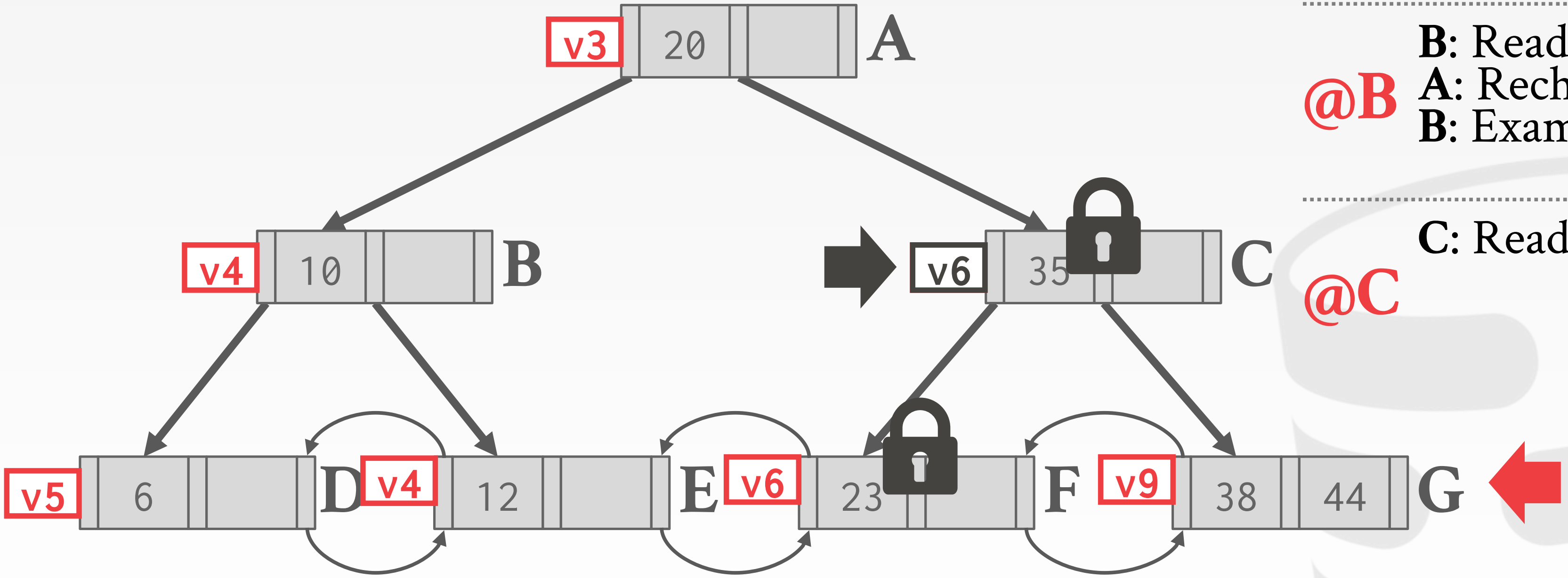


@A
A: Read v3
A: Examine Node

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

@C
C: Read v9

VERSIONED LATCHES: SEARCH 44

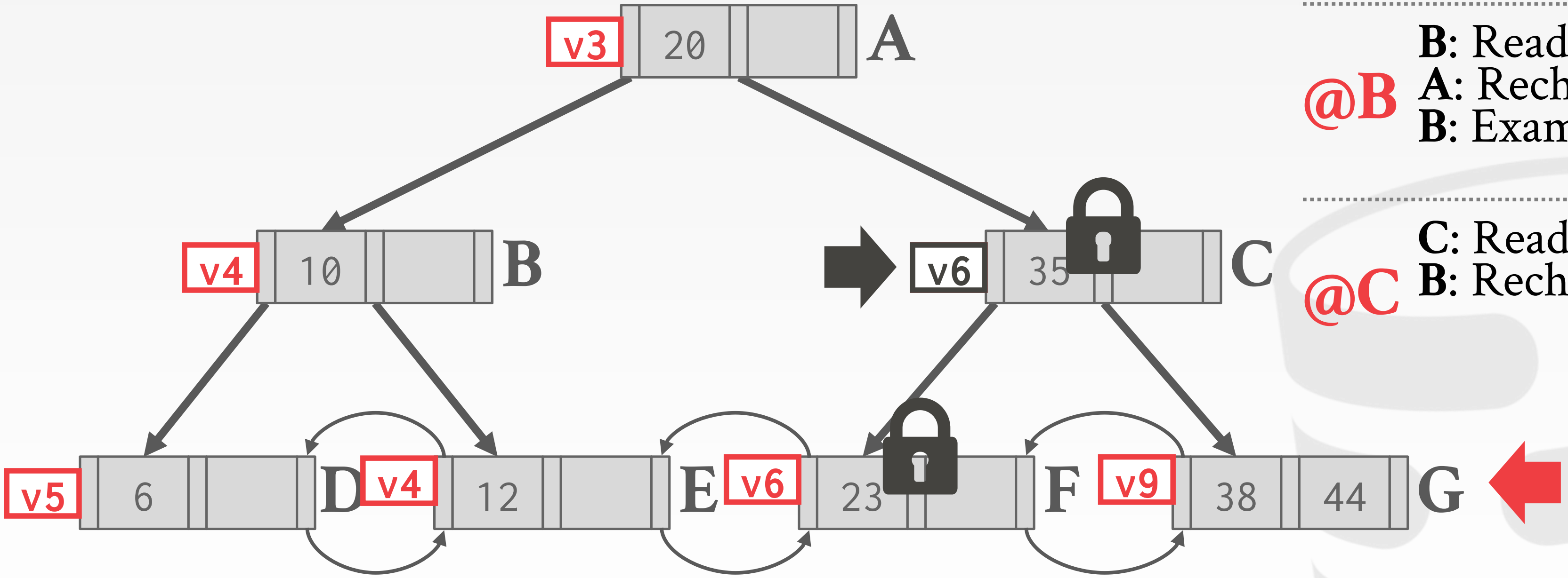


@A
A: Read v3
A: Examine Node

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

@C
C: Read v9

VERSIONED LATCHES: SEARCH 44

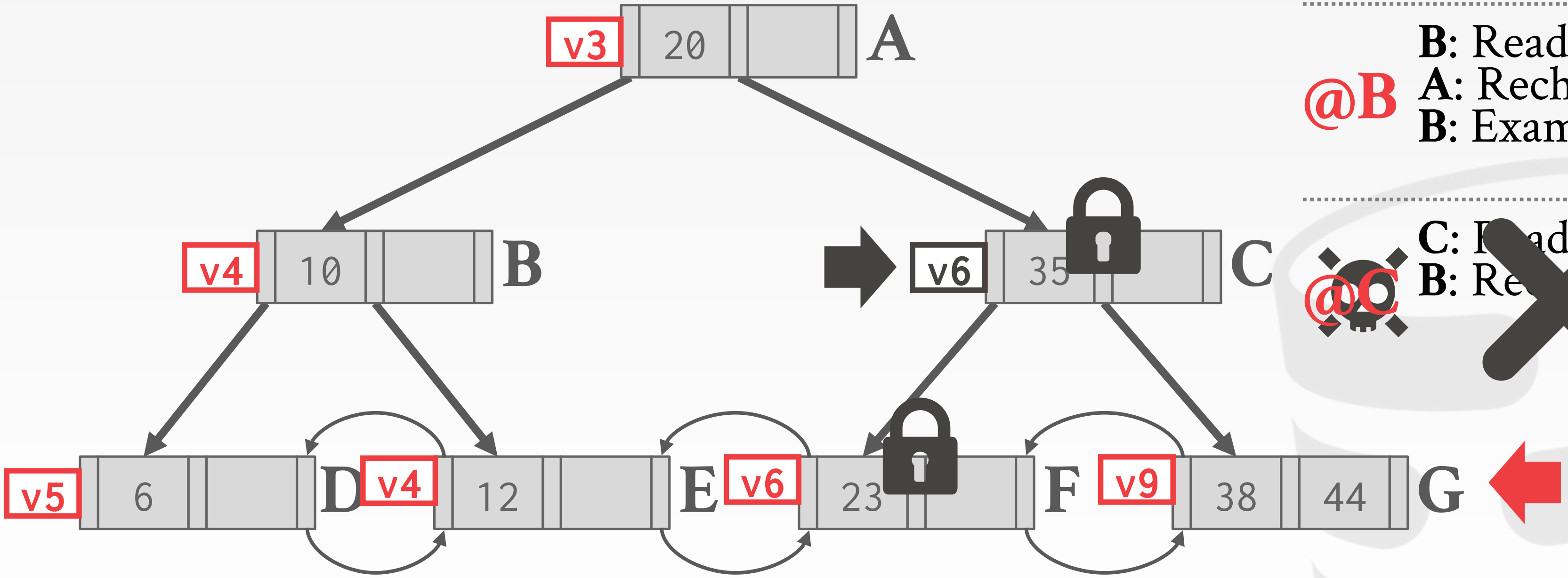


@A
A: Read v3
A: Examine Node

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

@C
C: Read v9
B: Recheck v5

VERSIONED LATCHES: SEARCH 44



@A
A: Read v3
A: Examine Node

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

@C
C: Read
B: Recheck v5

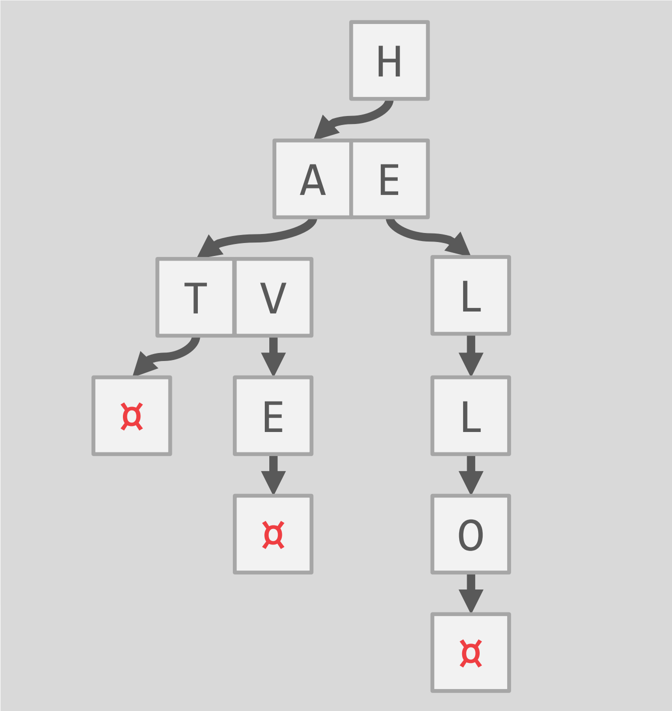
OBSERVATION

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

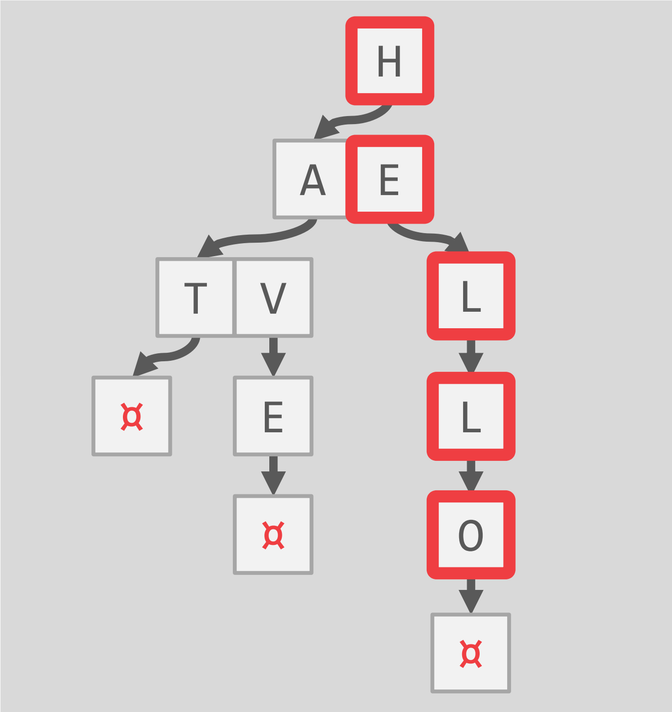


Use a digital representation of keys to examine prefixes one-by-one instead of comparing entire key.

→ 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.

TRIE KEY SPAN

The **span** of a trie level is the number of bits that each partial key / digit represents.

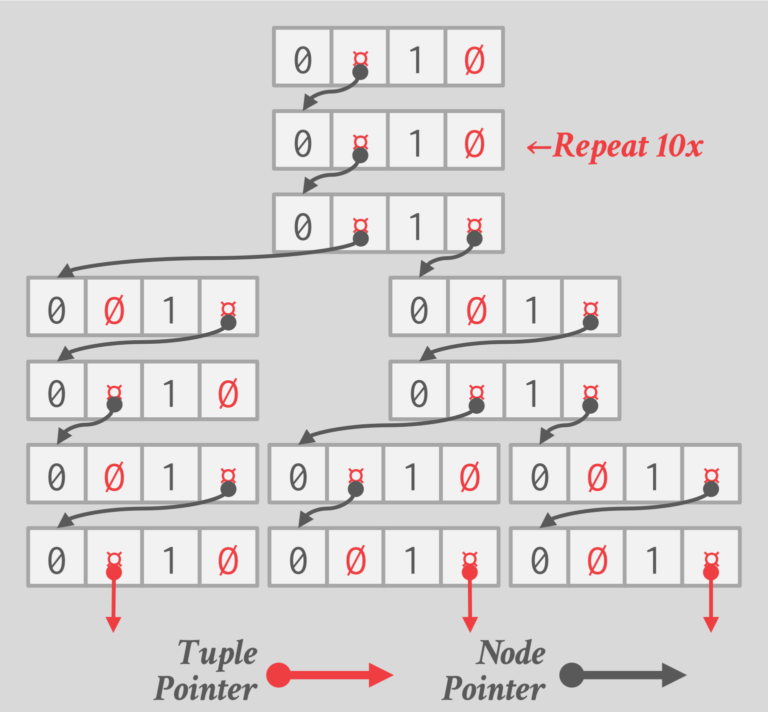
→ If the digit exists in the corpus, then store a pointer to the next level in the trie branch. Otherwise, store null.

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

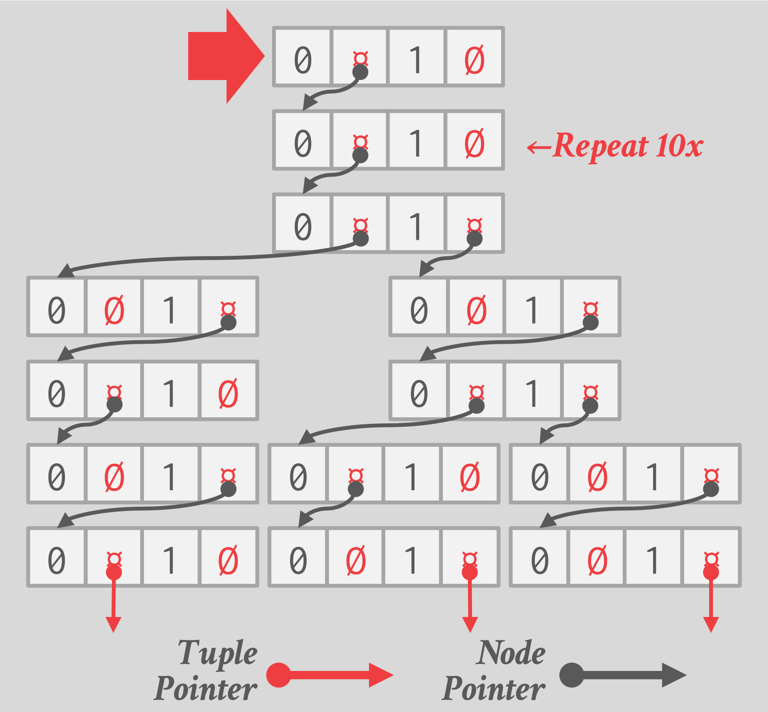


Keys: K10,K25,K31

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

TRIE KEY SPAN

1-bit Span Trie

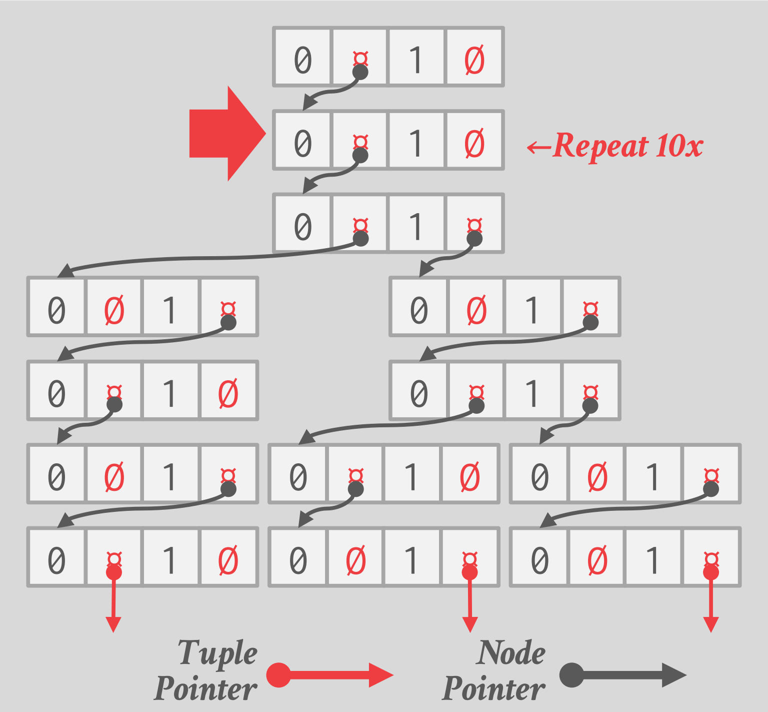


Keys: K10,K25,K31

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

TRIE KEY SPAN

1-bit Span Trie



Keys: K10,K25,K31

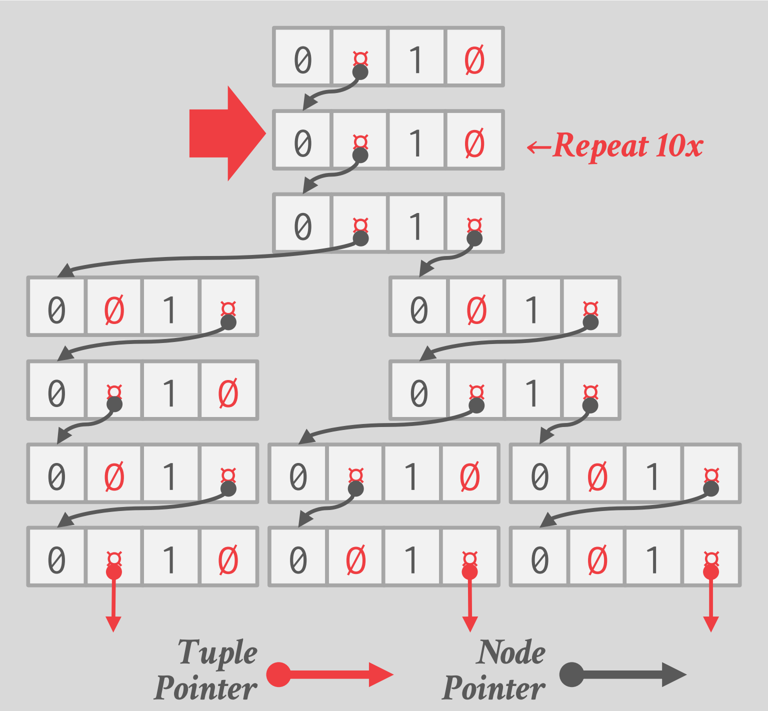
K10→ 00000000 00001010

K25→ 00000000 00011001

K31→ 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie

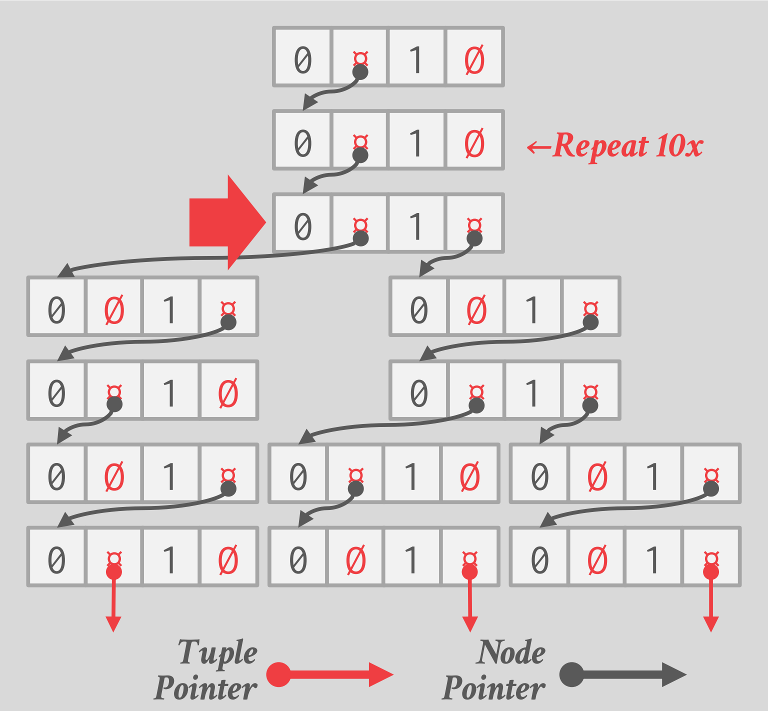


Keys: K10,K25,K31

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

TRIE KEY SPAN

1-bit Span Trie

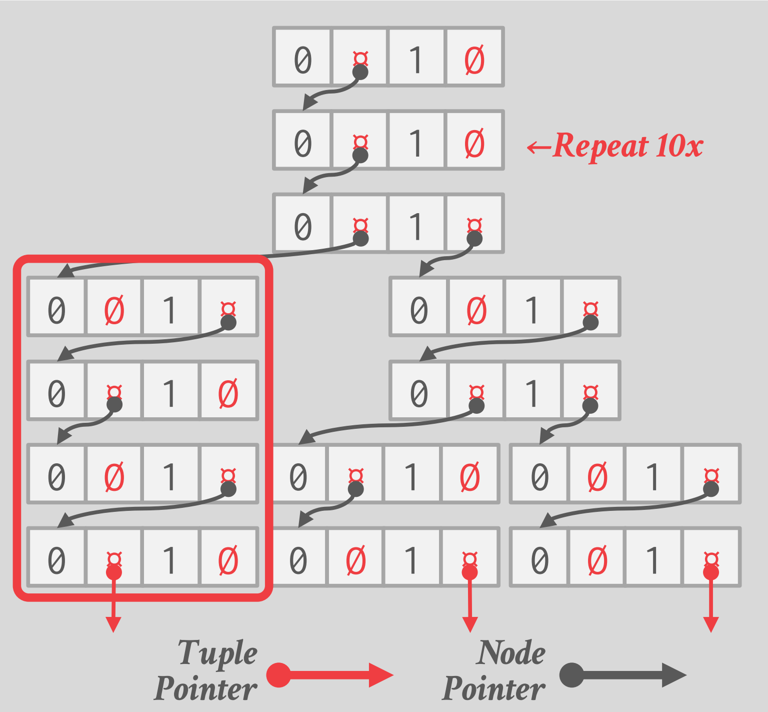


Keys: K10,K25,K31

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

TRIE KEY SPAN

1-bit Span Trie



Keys: K10,K25,K31

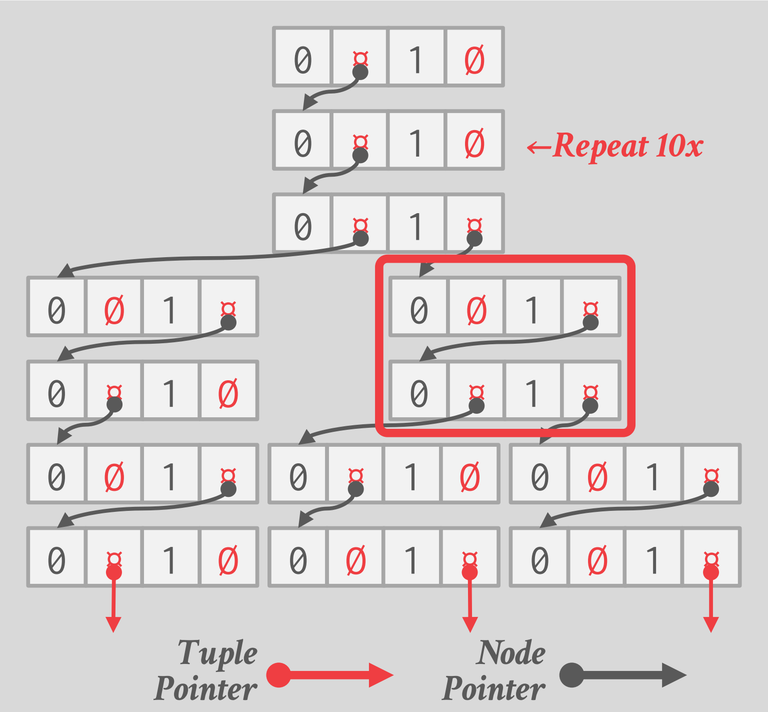
K10→ 00000000 0000$\boxed{1010}$

K25→ 00000000 00011001

K31→ 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie



Keys: K10,K25,K31

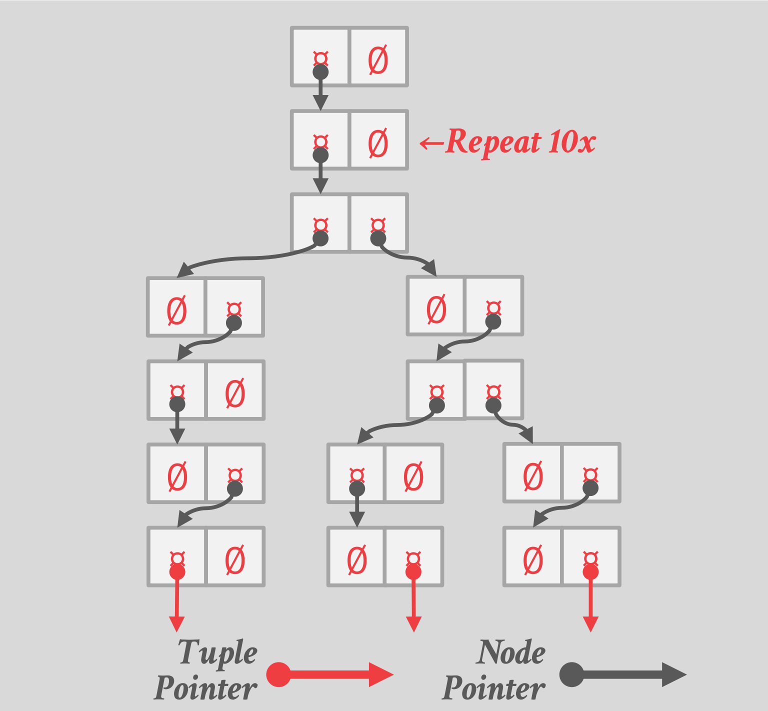
K10→ 00000000 00001010

K25→ 00000000 00011001

K31→ 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie

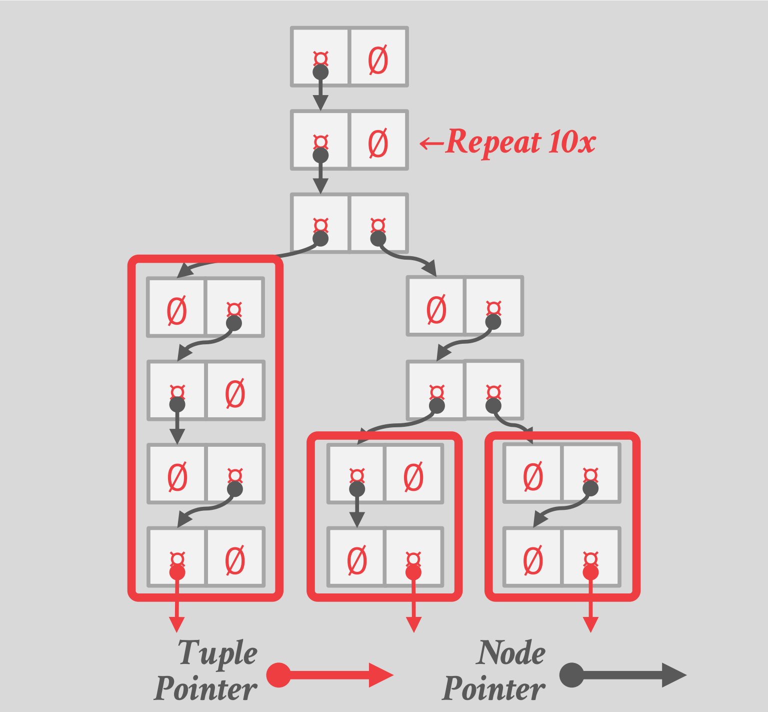


Keys: K10,K25,K31

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

TRIE KEY SPAN

1-bit Span Trie

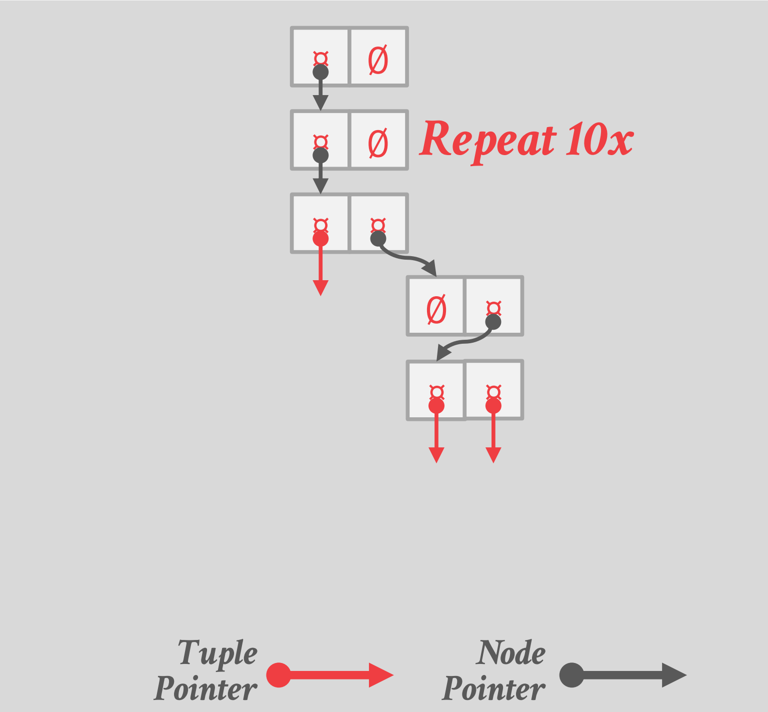


Keys: K10,K25,K31

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

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

A COMPARISON OF ADAPTIVE RADIX TREES AND HASH TABLES
ICDE 2015

JUDY ARRAYS: NODE TYPES

Every node can store up to 256 digits.

Not all nodes will be 100% full though.

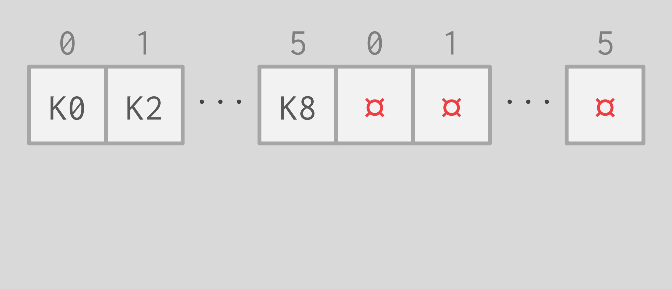
Adapt node's organization based on its keys.

→ **Linear Node:** Sparse Populations
→ **Bitmap Node:** Typical Populations
→ **Uncompressed Node:** Dense Population

A COMPARISON OF ADAPTIVE RADIX TREES AND HASH TABLES
ICDE 2015

JUDY ARRAYS: LINEAR NODES

Linear Node

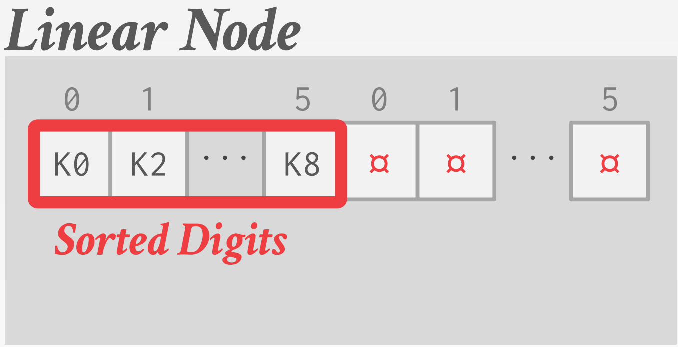


Store sorted list of partial prefixes up to <u>two</u> cache lines.

→ Original spec was one cache line

Store separate array of pointers to children ordered according to prefix sorted.

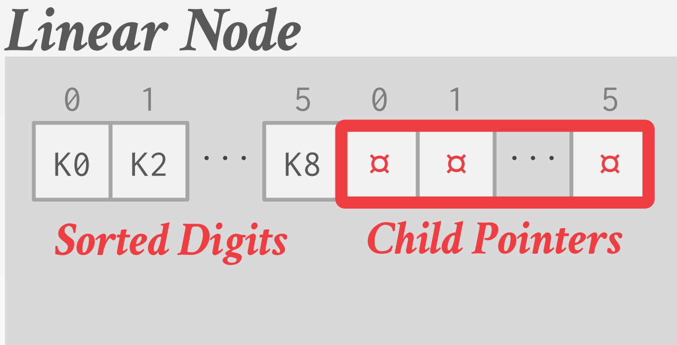
JUDY ARRAYS: LINEAR NODES



Store sorted list of partial prefixes up to <u>two</u> cache lines.
→ Original spec was one cache line

Store separate array of pointers to children ordered according to prefix sorted.

JUDY ARRAYS: LINEAR NODES



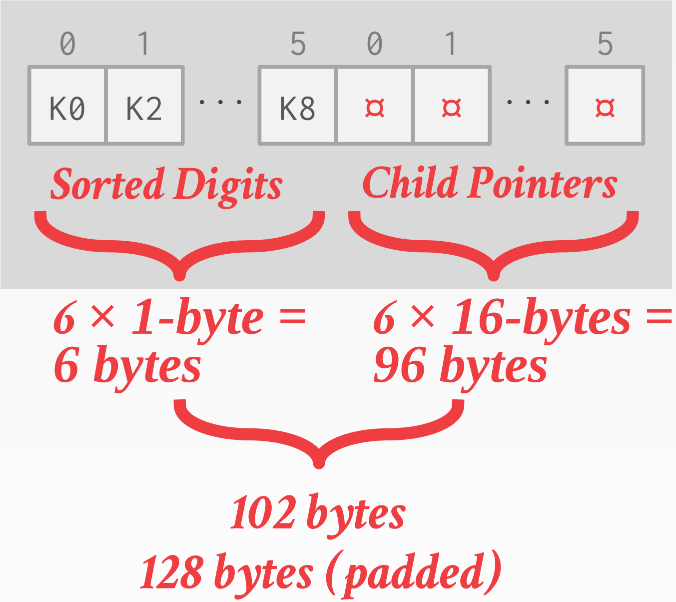
Store sorted list of partial prefixes up to two cache lines.

→ Original spec was one cache line

Store separate array of pointers to children ordered according to prefix sorted.

JUDY ARRAYS: LINEAR NODES

Linear Node

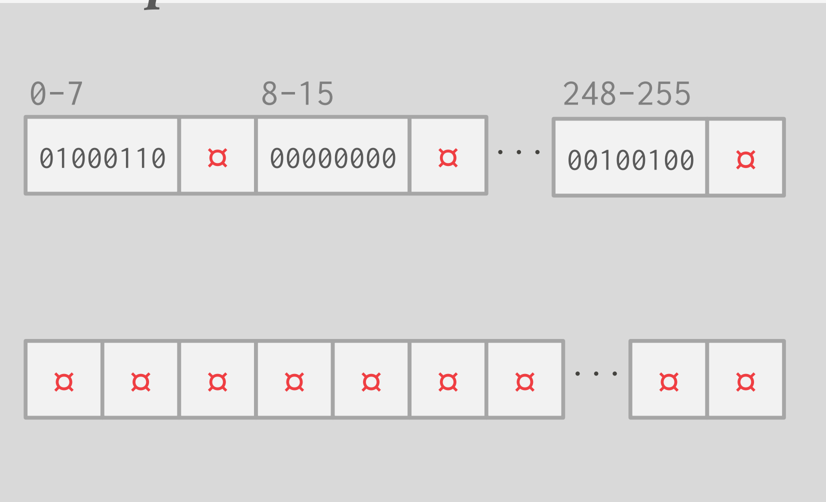


Store sorted list of partial prefixes up to <u>two</u> cache lines.
→ Original spec was one cache line

Store separate array of pointers to children ordered according to prefix sorted.

JUDY ARRAYS: BITMAP NODES

Bitmap Node

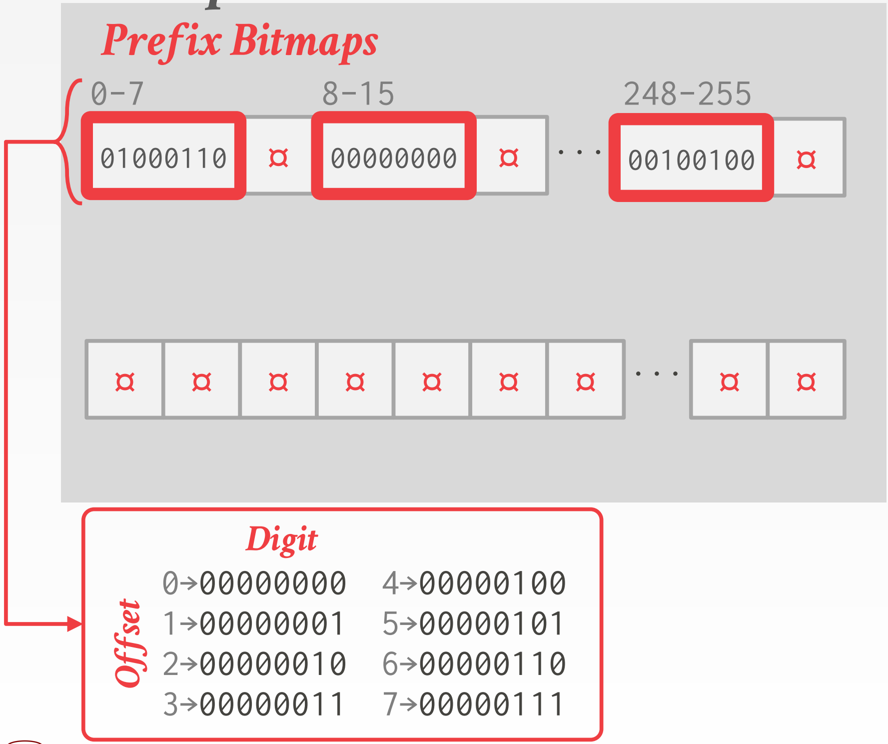


256-bit map to mark whether a prefix is present in node.

Bitmap is divided into eight segments, each with a pointer to a sub-array with pointers to child nodes.

JUDY ARRAYS: BITMAP NODES

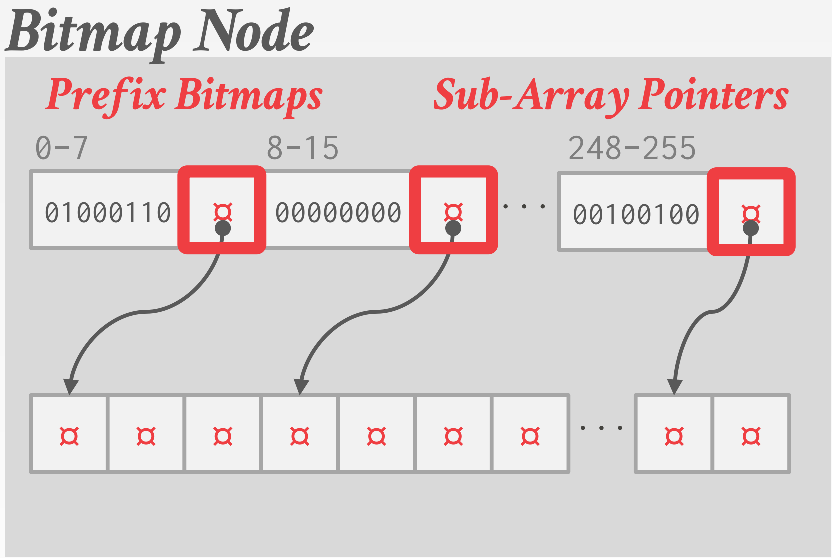
Bitmap Node



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

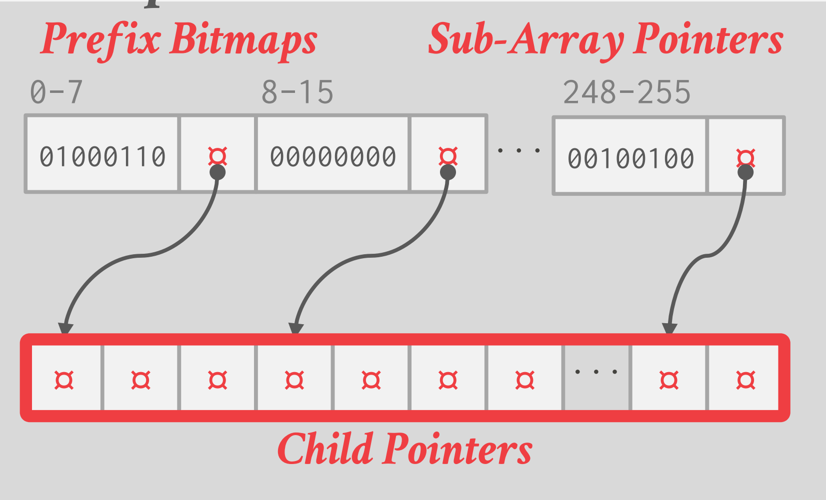


256-bit map to mark whether a prefix is present in node.

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



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.

ADAPATIVE 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.

THE ADAPTIVE RADIX TREE: ARTFUL INDEXING FOR MAIN-MEMORY DATABASES
ICDE 2013

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.

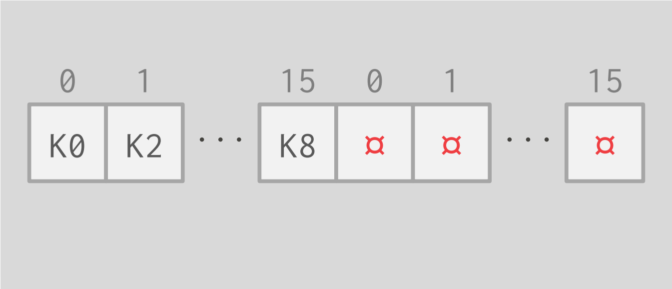
Difference #2: Purpose

→ 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

Node16

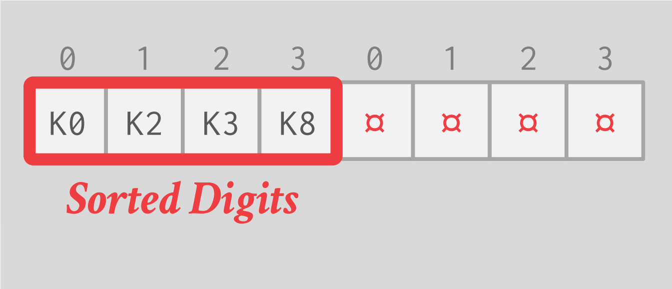


Store only the 8-bit digits that exist at a given node in a sorted array.

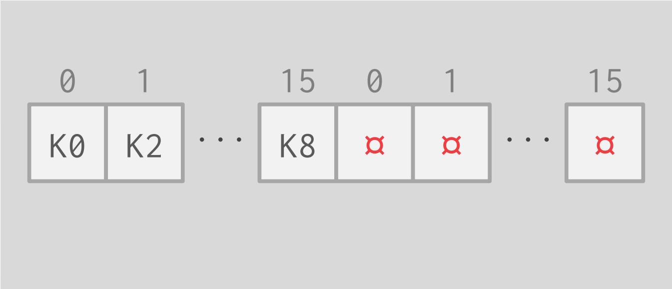
The offset in sorted digit array corresponds to offset in value array.

ART: INNER NODE TYPES (1)

Node4



Node16



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)

Node4

0	1	2	3	0	1	2	3
K0	K2	K3	K8	¤	¤	¤	¤

Sorted Digits ***Child Pointers***

Node16

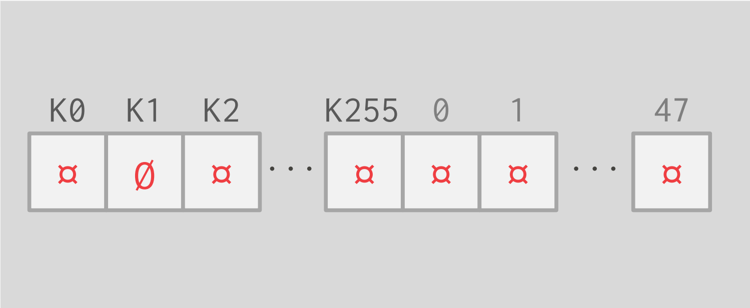
0	1		15	0	1		15
K0	K2	···	K8	¤	¤	···	¤

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 (2)

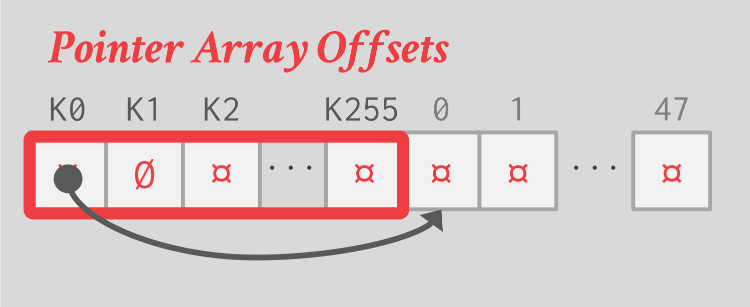
Node48



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)

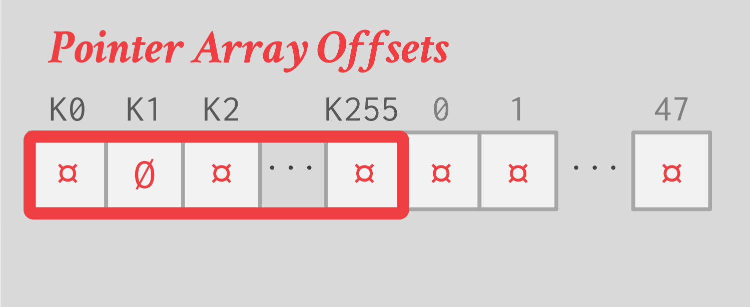
Node48



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)

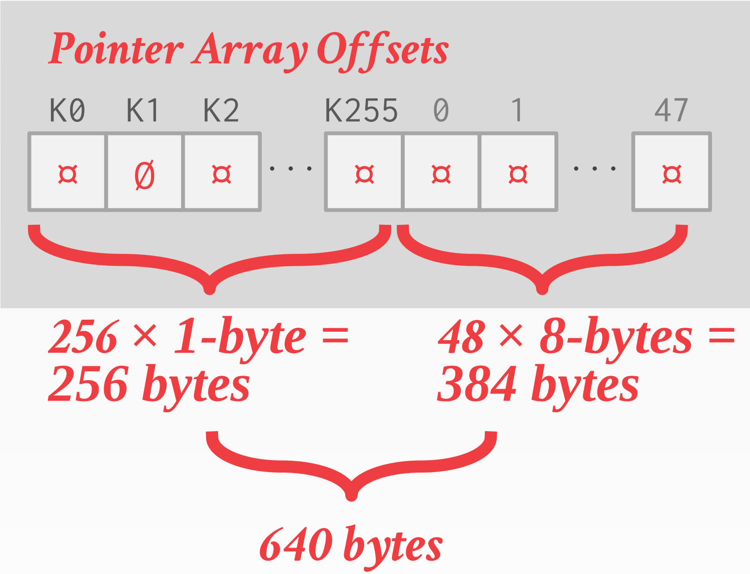
Node48



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)

Node48



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

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

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.

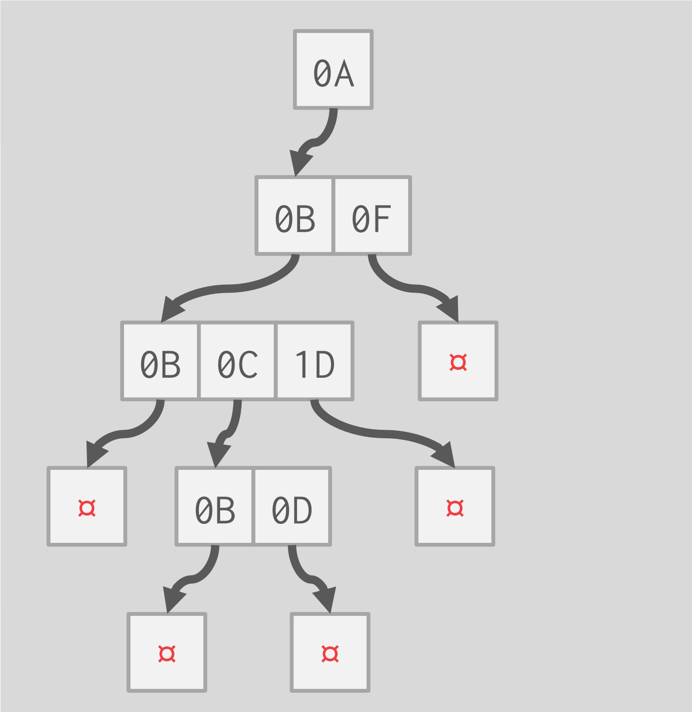
ART: BINARY COMPARABLE KEYS

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.

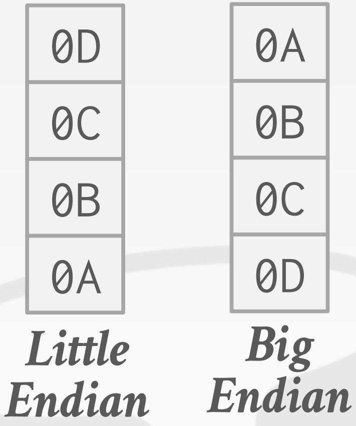
ART: BINARY COMPARABLE KEYS

8-bit Span Radix Tree



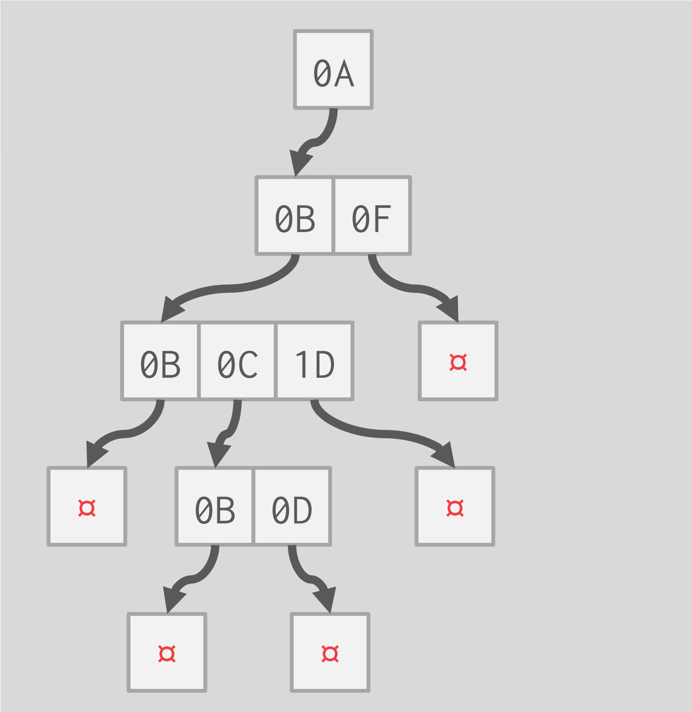
Int Key: 168496141

Hex Key: 0A 0B 0C 0D



ART: BINARY COMPARABLE KEYS

8-bit Span Radix Tree

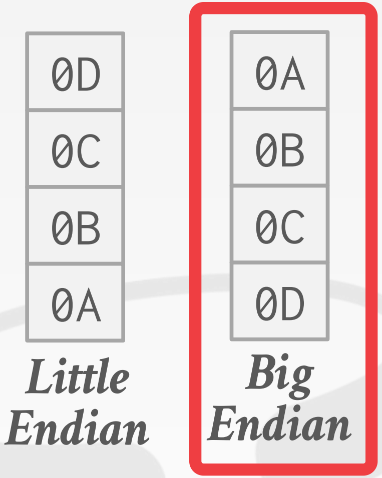


Int Key: 168496141

Hex Key: 0A 0B 0C 0D

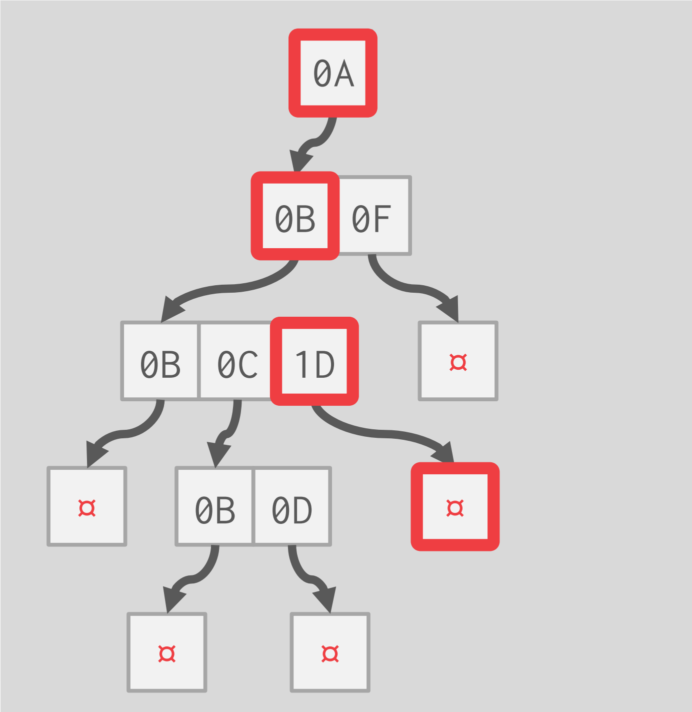
Find: 658205

Hex: 0A 0B 1D



ART: BINARY COMPARABLE KEYS

8-bit Span Radix Tree

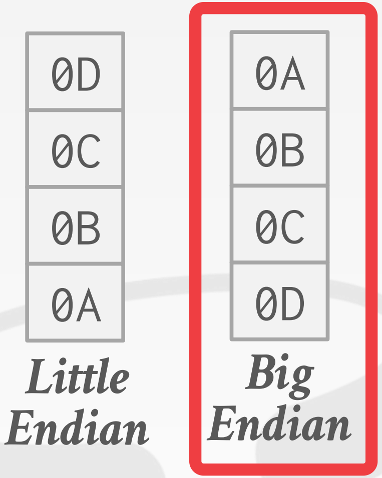


Int Key: 168496141

Hex Key: 0A 0B 0C 0D

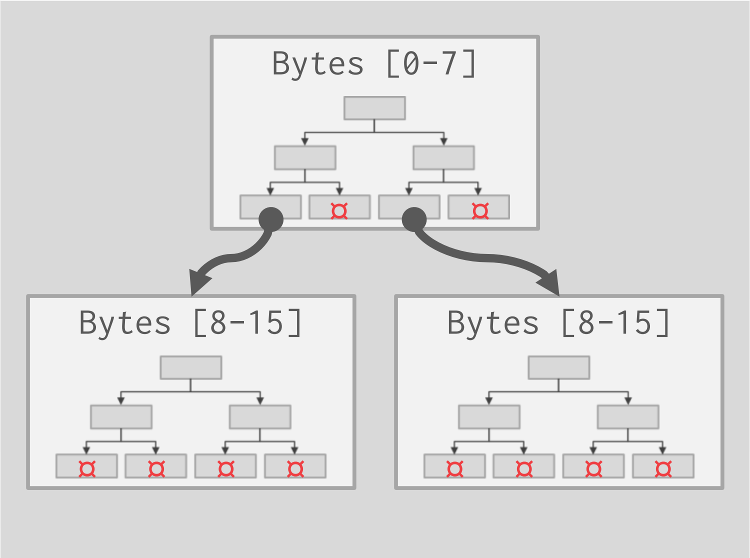
Find: 658205

Hex: 0A 0B 1D



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.

CACHE CRAFTINESS FOR FAST MULTICORE KEY-VALUE STORAGE
EUROSYS 2012

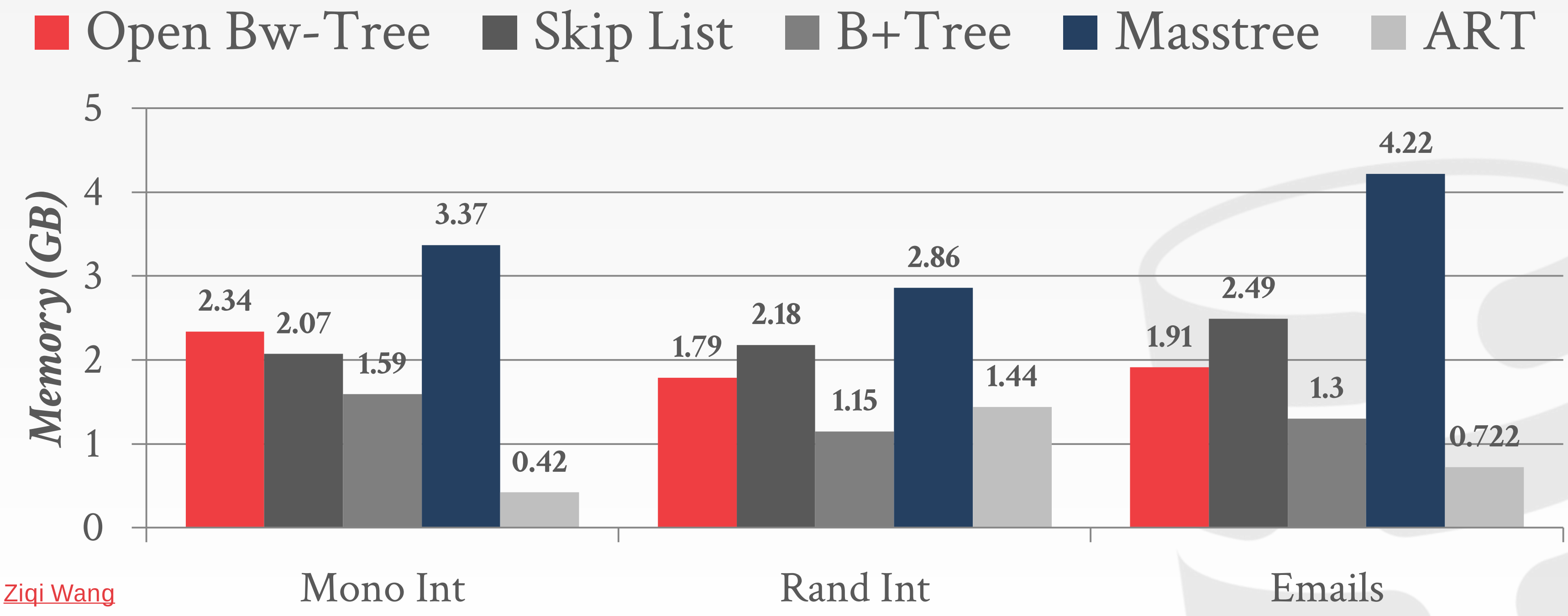
IN-MEMORY INDEXES

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

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