Intro to Database Systems (15-445/645)

09 Index Concurrency Control

Carnegie Mellon University

SPRING 2023

Charlie Garrod

ADMINISTRIVIA

Homework 2 due February 17th.

Project 1 due February 19th.
→ Saturday office hours: February 18th 3-5 p.m.

ADMINISTRIVI

om February 17th.

ry 19th.

ru

You're the page I'd never evict from my buffer pool.

You won't need NoSQL after we join all our relations.

Can I be your lock manager? So that I can grant you an exclusive lock to my heart.

15-445 / 15-645 PARTICIPATION QUIZ

For a cuckoo hashing scheme with 1000 buckets, 2 hash functions, and 4 slots per bucket: In the worst-case scenario, what is the minimum number of insertions (into an initially empty table) that might require the table to be rehashed?

https://bit.ly/cmu-db-quiz

LAST TIME

B+Trees
→ Use in a DBMS
→ Design Choices
→ Optimizations

LEAF NODE VALUES

Approach #1: Record IDs

→ A pointer to the location of the tuple to which the index entry corresponds.

Approach #2: Tuple Data

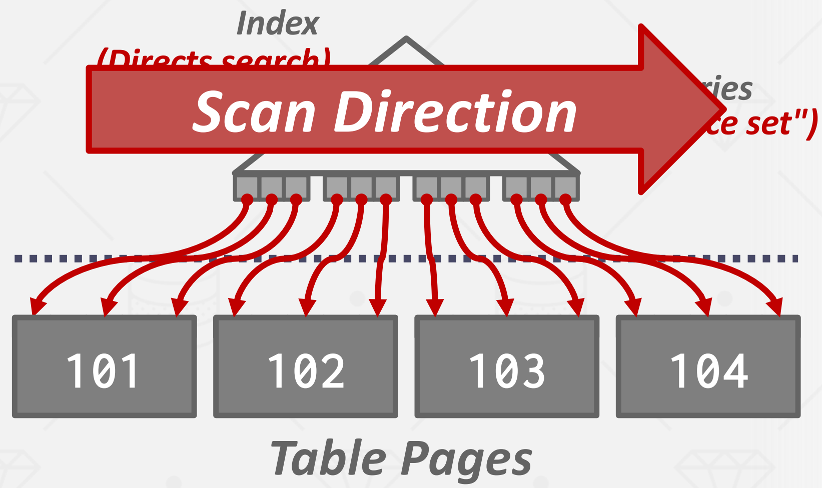
→ The leaf nodes store the actual contents of the tuple.

→ Secondary indexes must store the Record ID as their values.

CLUSTERED B+TREE

Traverse to the left-most leaf page and then retrieve tuples from all leaf pages.

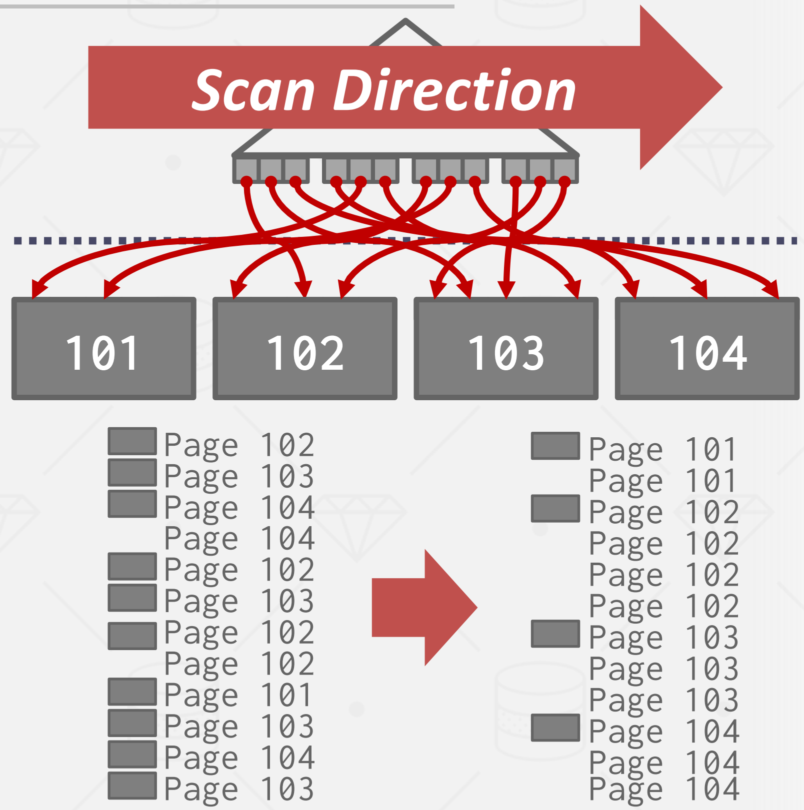
This will always be better than sorting data for each query.



INDEX SCAN PAGE SORTING

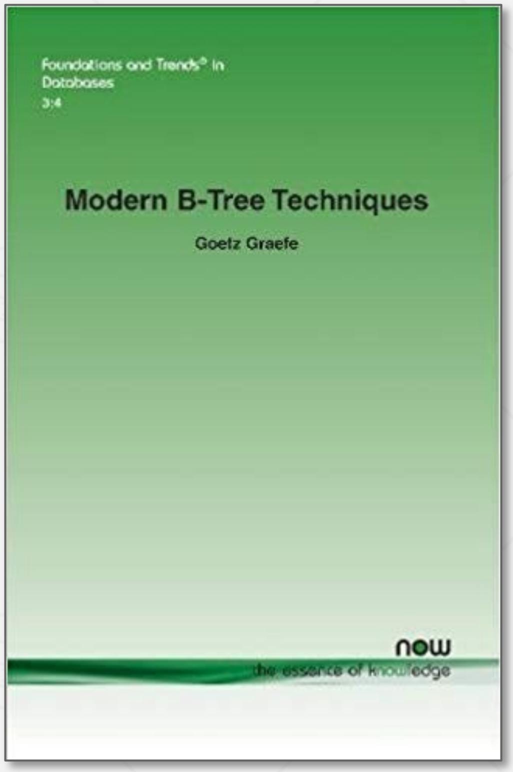
Retrieving tuples in the order they appear in a non-clustered index is inefficient due to redundant reads.

The DBMS can first figure out all the tuples that it needs and then sort them based on their Page ID.



B+TREE DESIGN CHOICES

Node Size
Merge Threshold
Variable-Length Keys
Intra-Node Search



TODAY

Finish B+Tree Design and Optimization
Index Concurrency Control

INTRA-NODE SEARCH

Approach #1: Linear

→ Scan node keys from beginning to end.

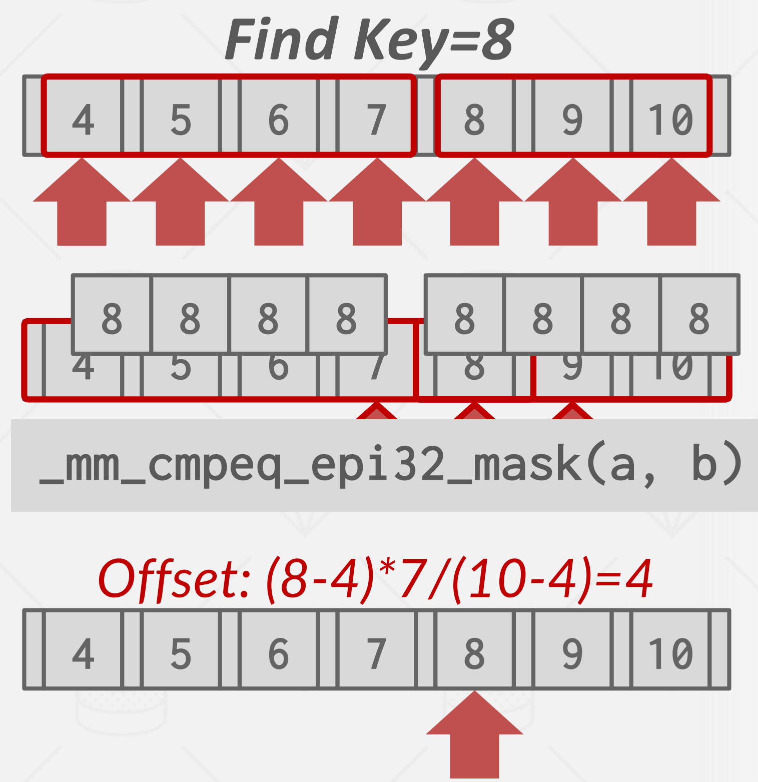
→ Use SIMD to vectorize comparisons.

Approach #2: Binary

→ Jump to middle key, pivot left/right depending on comparison.

Approach #3: Interpolation

→ Approximate location of desired key based on known distribution of keys.



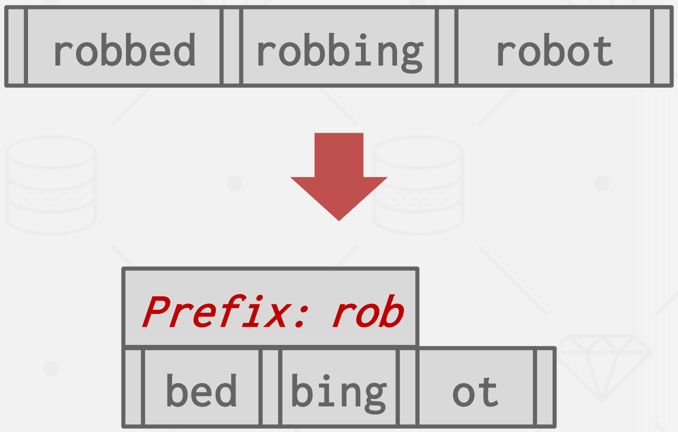
OPTIMIZATIONS

- Prefix Compression
- Deduplication
- Suffix Truncation
- Pointer Swizzling
- Bulk Insert
- Buffer Updates
- Many more…

PREFIX COMPRESSION

Sorted keys in the same leaf node are likely to have the same prefix.

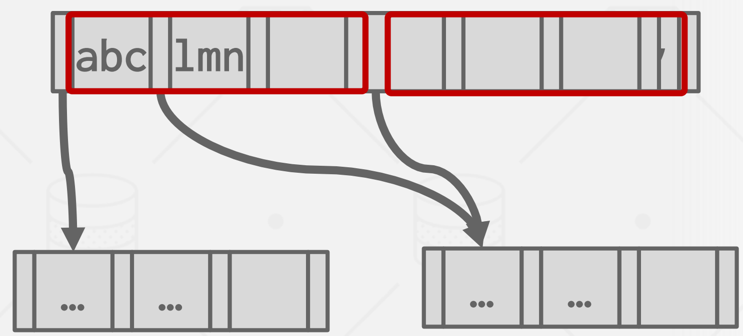
Instead of storing the entire key each time, extract common prefix and store only unique suffix for each key.
→ Many variations.



SUFFIX TRUNCATION

The keys in the inner nodes are only used to "direct traffic".
→ We don't need the entire key.

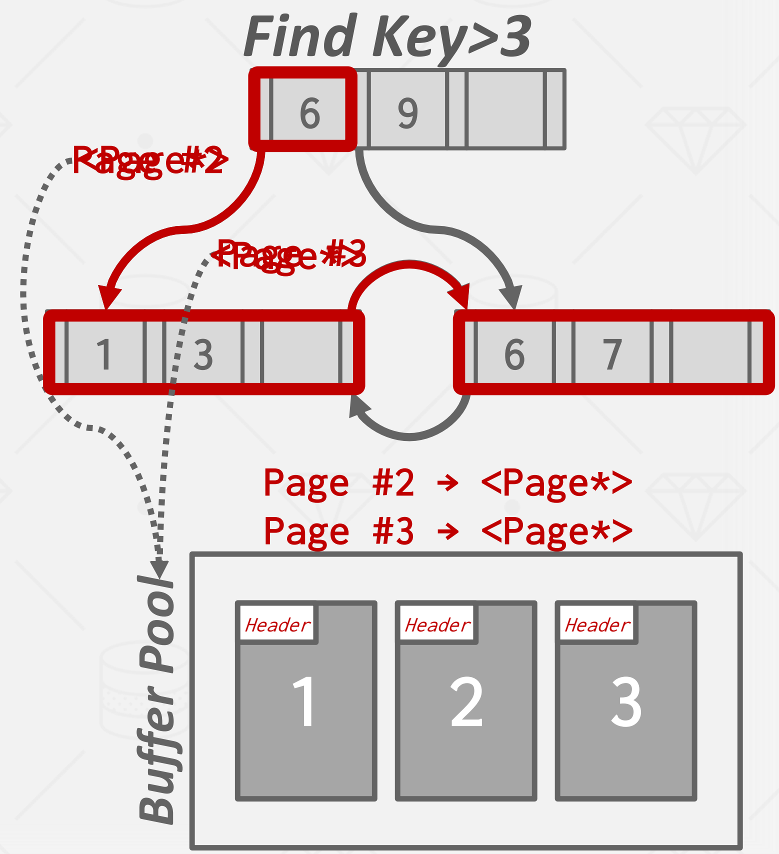
Store a minimum prefix that is needed to correctly route probes into the index.



POINTER SWIZZLING

Nodes use page ids to reference other nodes in the index. The DBMS must get the memory location from the page table during traversal.

If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.

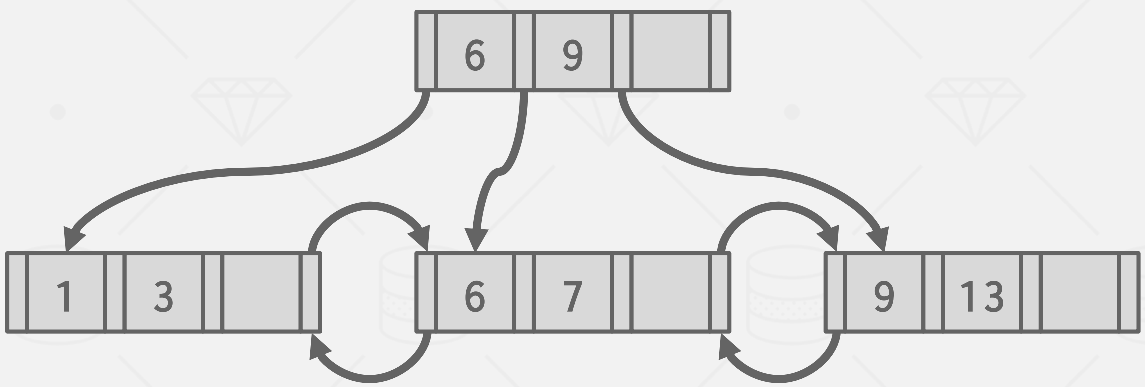


BULK INSERT

The fastest way to build a new B+Tree for an existing table is to first sort the keys and then build the index from the bottom up.

Keys: 3, 7, 9, 13, 6, 1

Sorted Keys: 1, 3, 6, 7, 9, 13



B+TREE CONCLUSION

The venerable B+Tree is (almost) always a good choice for your DBMS.

TODAY

Finish B+Tree Design and Optimization
Index Concurrency Control

OBSERVATION

We (mostly) assumed all the data structures that we have discussed so far are single-threaded.

But a DBMS needs to allow multiple threads to safely access data structures to take advantage of additional CPU cores and hide disk I/O stalls.

They Don't Do This!

VOLTDB

redis

H-Store

CONCURRENCY CONTROL

A **concurrency control** protocol is the method that the DBMS uses to ensure "correct" results for concurrent operations on a shared object.

A protocol's correctness criteria can vary:
→ **Logical Correctness:** Can a thread see the data that it is supposed to see?
→ **Physical Correctness:** Is the internal representation of the object sound?

TODAY'S AGENDA

Latches Overview
Hash Table Latching
B+Tree Latching
Leaf Node Scans

LOCKS VS. LATCHES

Locks

→ Protect the database's logical contents from other txns.
→ Held for txn duration.
→ Need to be able to rollback changes.

Latches

→ Protect the critical sections of the DBMS's internal data structure from other threads.
→ Held for operation duration.
→ Do not need to be able to rollback changes.

LOCKS VS. LATCHES

Lecture #15

	Locks	*Latches*
Separate…	User Transactions	Threads
Protect…	Database Contents	In-Memory Data Structures
During…	Entire Transactions	Critical Sections
Modes…	Shared, Exclusive, Update, Intention	Read, Write
Deadlock	Detection & Resolution	Avoidance
…by…	Waits-for, Timeout, Aborts	Coding Discipline
Kept in…	Lock Manager	Protected Data Structure

Source: Goetz Graefe

LATCH MODES

Read Mode

→ Multiple threads can read the same object at the same time.
→ A thread can acquire the read latch if another thread has it in read mode.

Write Mode

→ Only one thread can access the object.
→ A thread cannot acquire a write latch if another thread has it in any mode.

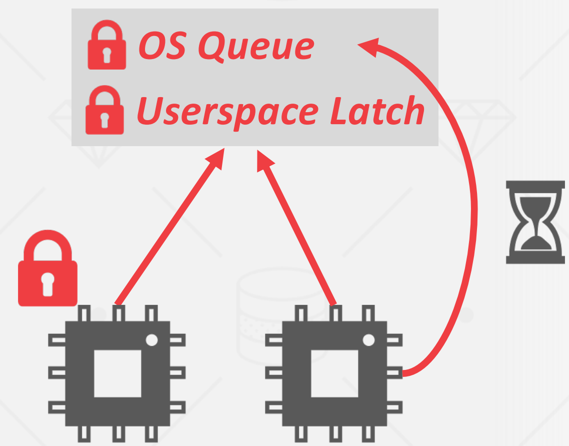
Compatibility Matrix

	Read	Write
Read	✓	X
Write	X	X

LATCH IMPLEMENTATIONS

Approach #1: Blocking OS Mutex
→ Simple to use
→ Non-scalable (about 25ns per lock/unlock invocation)
→ Example: `std::mutex` → `pthread_mutex` → `futex`

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



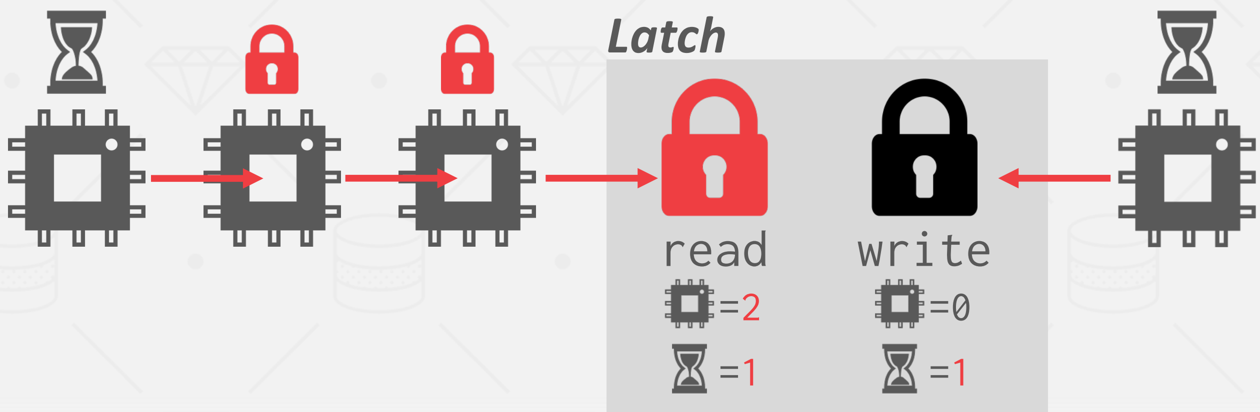
LATCH IMPLEMENTATIONS

Approach #2: Reader-Writer Latches

→ Allows for concurrent readers. Must manage read/write queues to avoid starvation.

→ Can be implemented on top of spinlocks.

→ Example: `std::shared_mutex` → `pthread_rwlock`



HASH TABLE LATCHING

Easy to support concurrent access due to the limited ways threads access the data structure.

→ All threads move in the same direction and only access a single page/slot at a time.
→ Deadlocks are not possible.

To resize the table, take a global write latch on the entire table (e.g., in the header page).

HASH TABLE LATCHING

Approach #1: Page Latches

→ Each page has its own reader-writer latch that protects its entire contents.

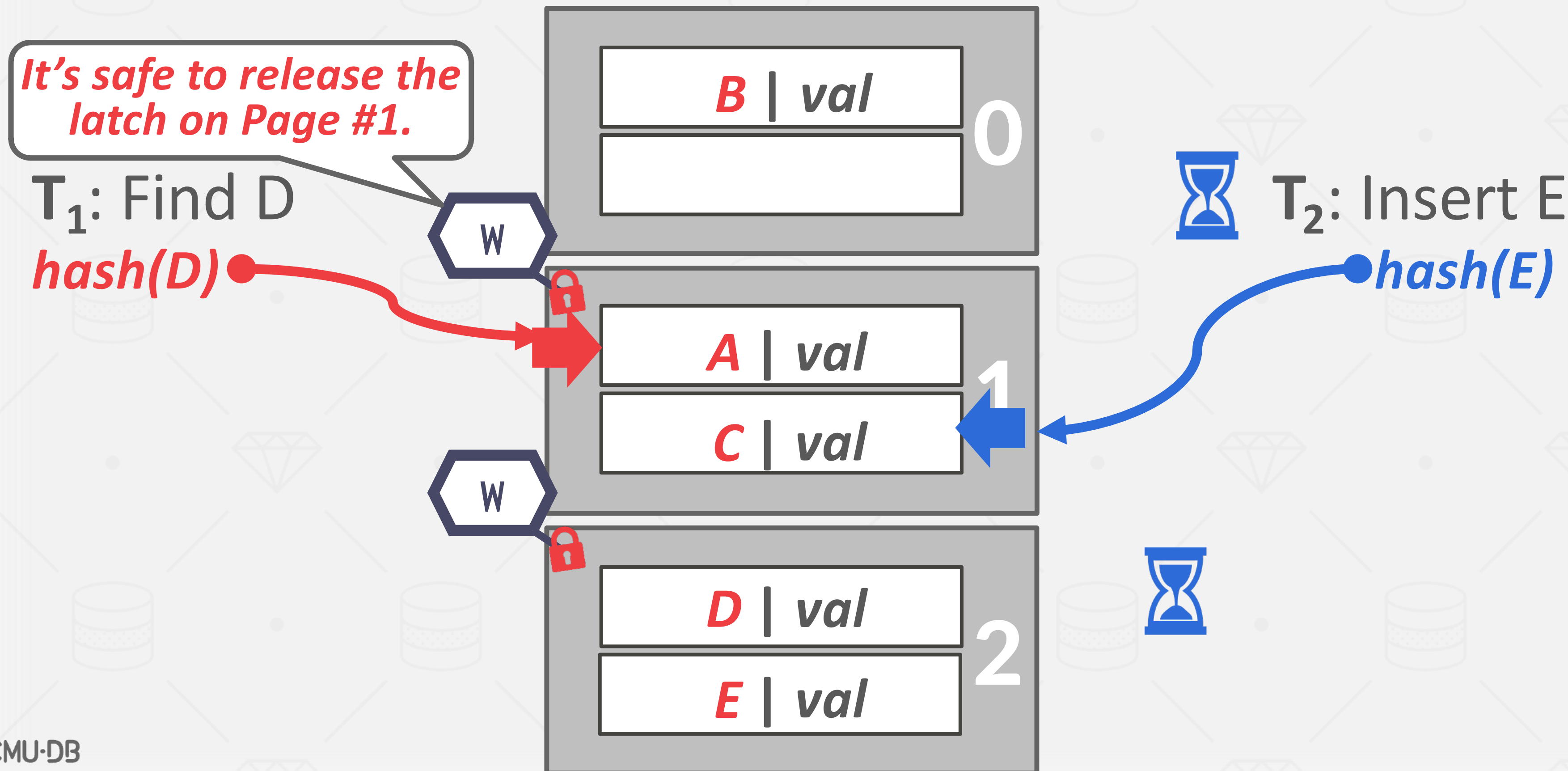
→ Threads acquire either a read or write latch before they access a page.

Approach #2: Slot Latches

→ Each slot has its own latch.

→ Can use a single-mode latch to reduce meta-data and computational overhead.

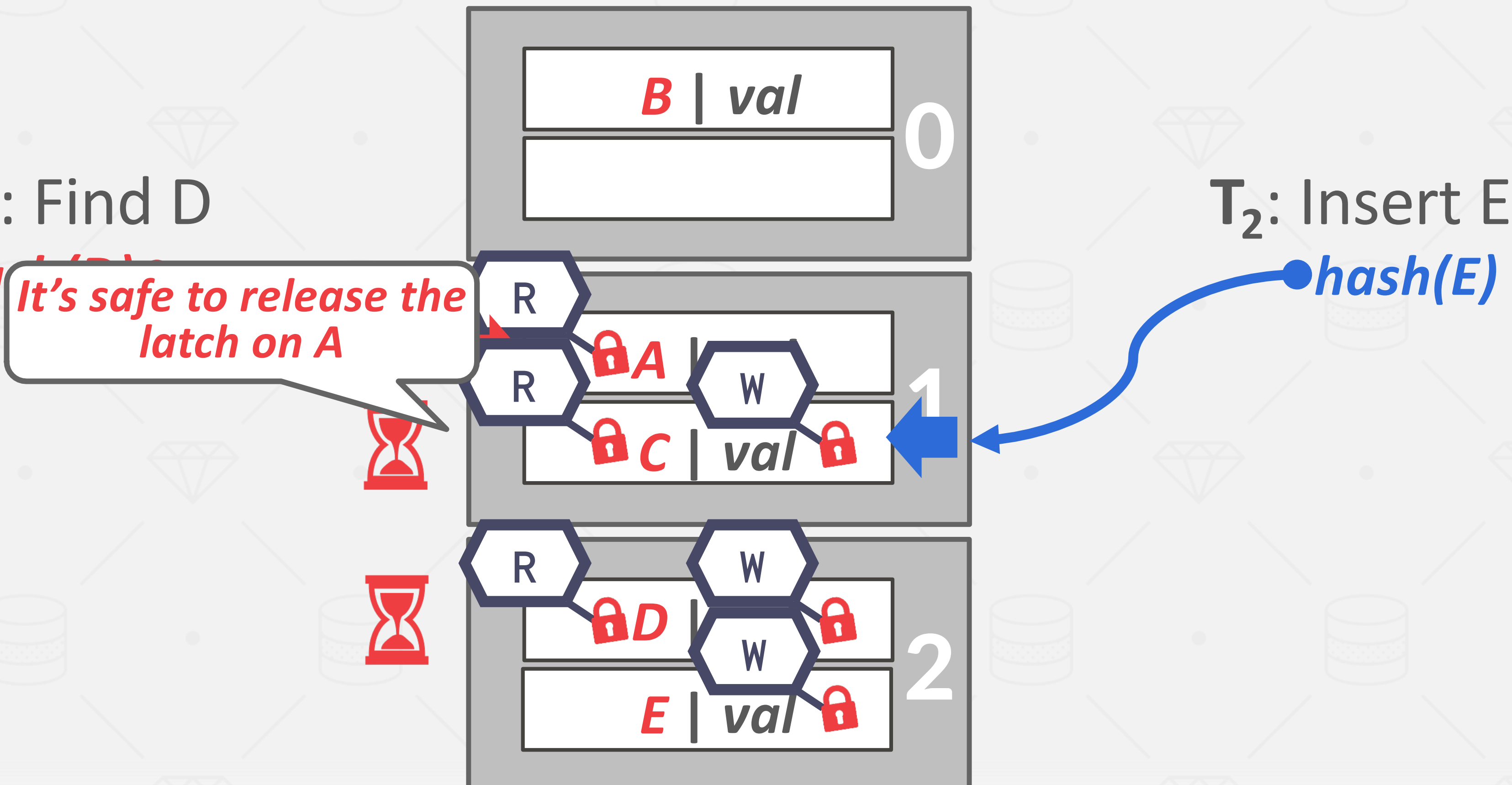
HASH TABLE – PAGE LATCHES



HASH TABLE – SLOT LATCHES

T_1: Find D

T_2: Insert E



B+TREE CONCURRENCY CONTROL

We want to allow multiple threads to read and update a B+Tree at the same time.

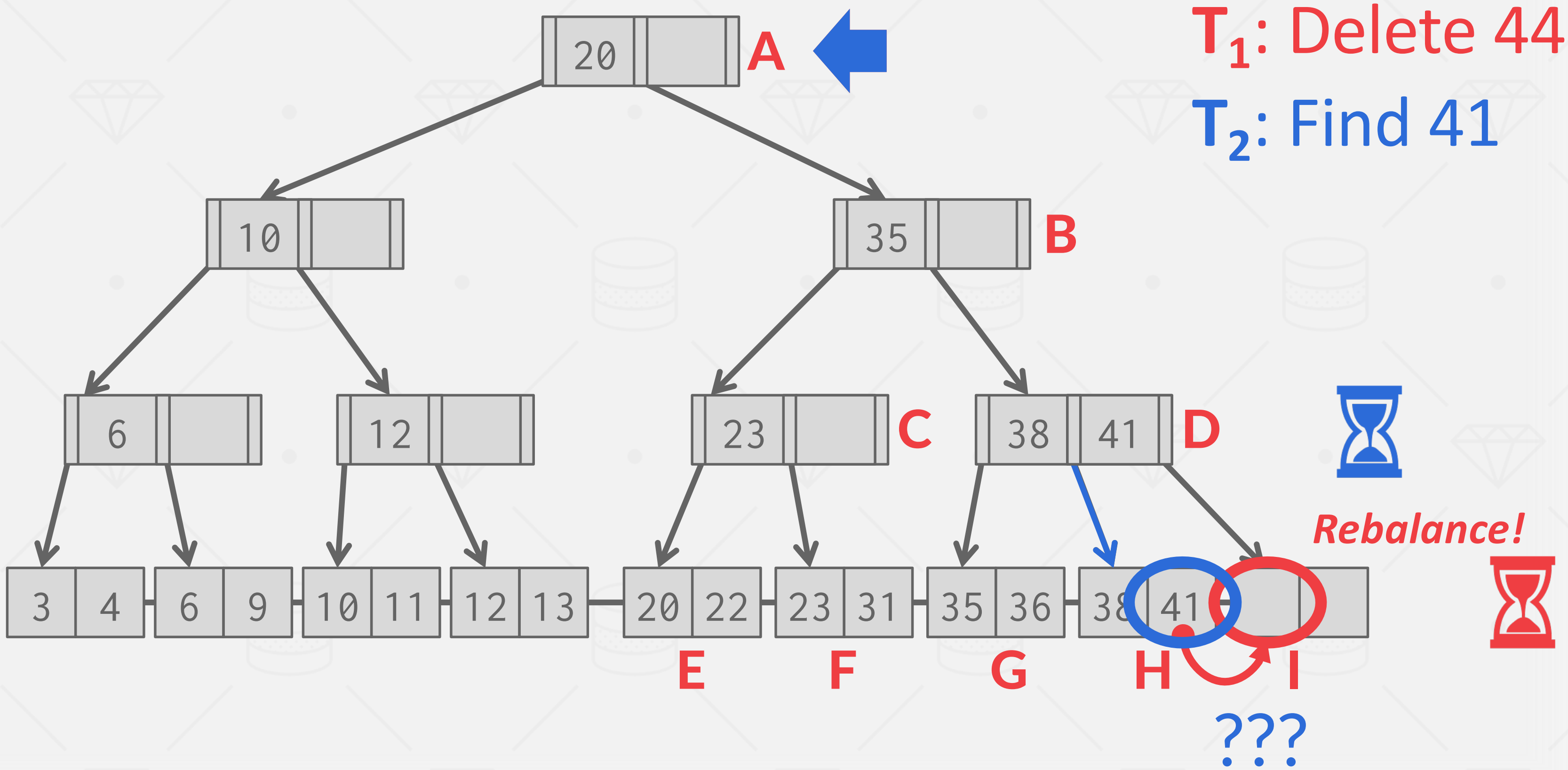
We need to protect against two types of problems:

→ Threads trying to modify the contents of a node at the same time.
→ One thread traversing the tree while another thread splits/merges nodes.

B+TREE MULTI-THREADED EXAMPLE

T_1: Delete 44

T_2: Find 41



LATCH CRABBING/COUPLING

Protocol to allow multiple threads to access/modify B+Tree at the same time.

→ Get latch for parent
→ Get latch for child
→ Release latch for parent if "safe"

A **safe node** is one that will not split or merge when updated.

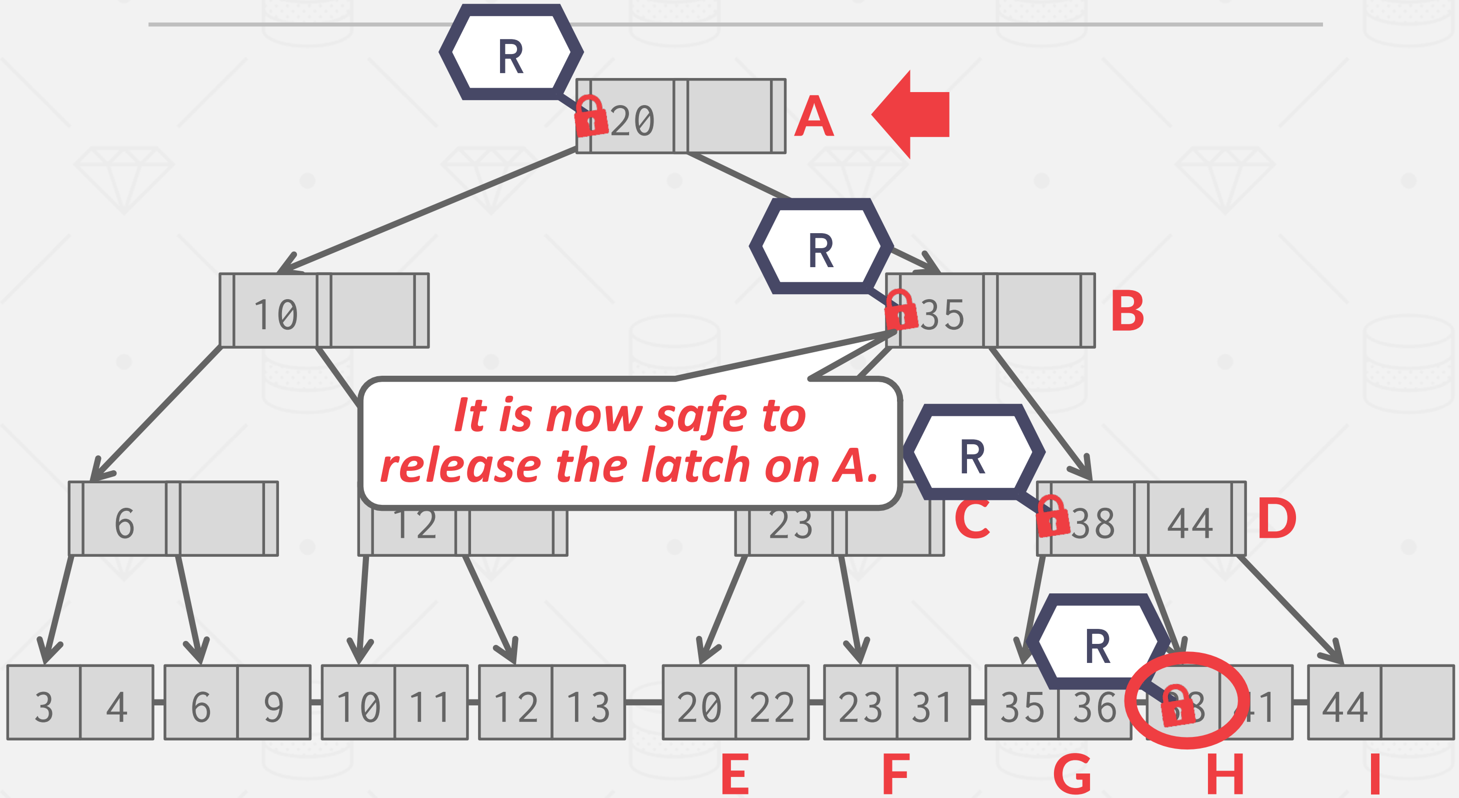
→ Not full (on insertion)
→ More than half-full (on deletion)

LATCH CRABBING/COUPLING

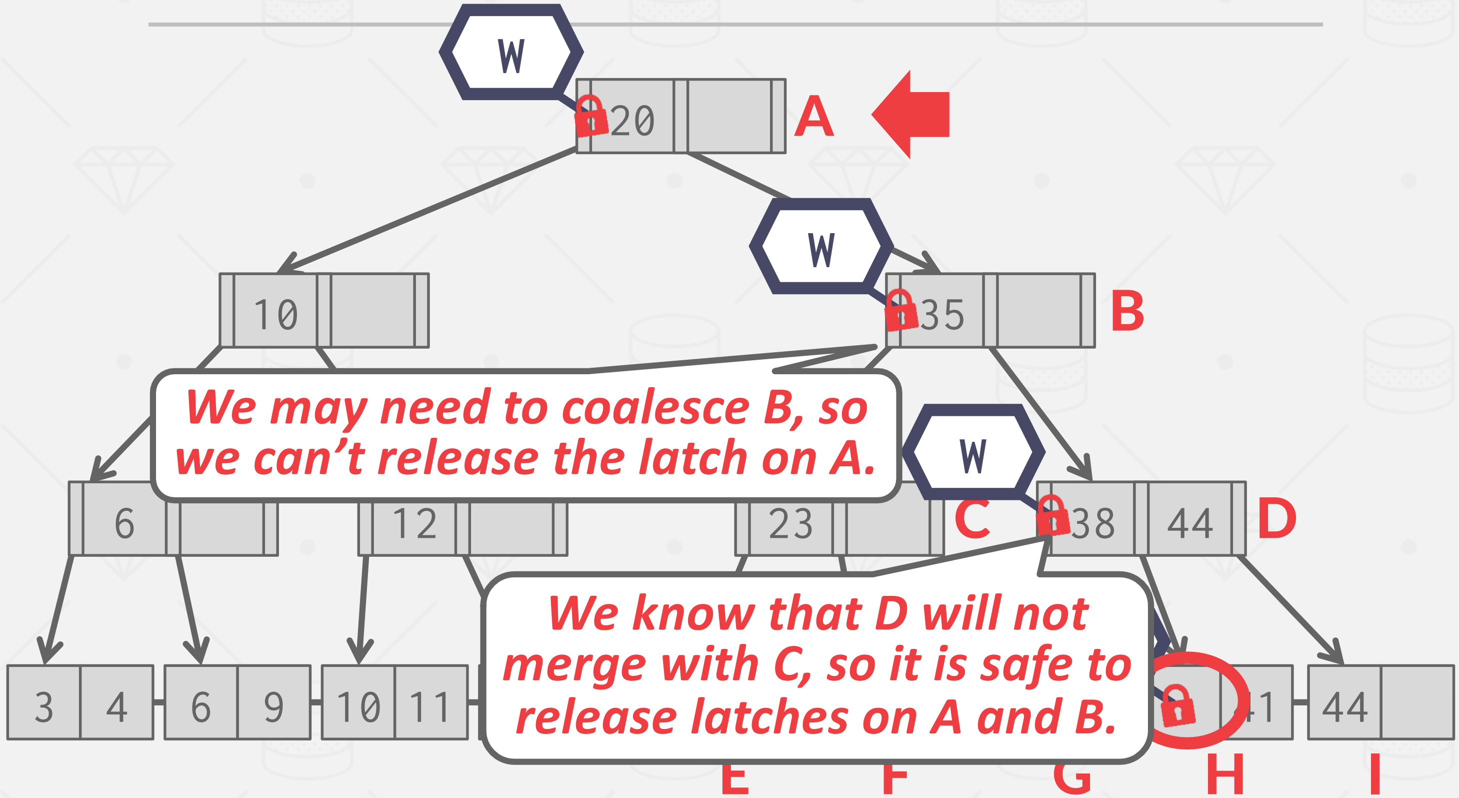
Find: Start at root and traverse down the tree:
→ Acquire R latch on child,
→ Then unlatch parent.
→ Repeat until we reach the leaf node.

Insert/Delete: Start at root and go down, obtaining W latches as needed. Once child is latched, check if it is safe:
→ If child is safe, release all latches on ancestors

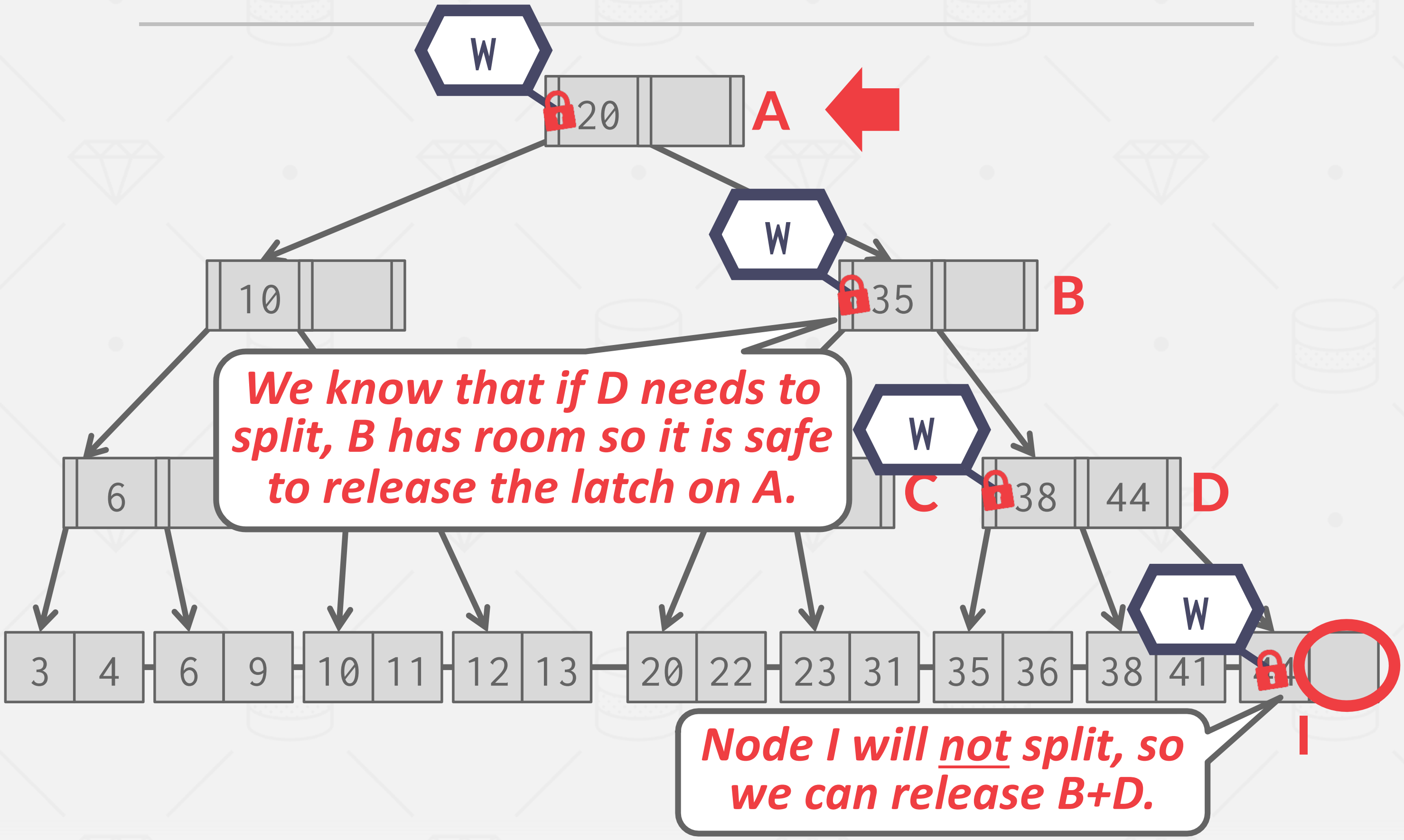
EXAMPLE #1 – FIND 38



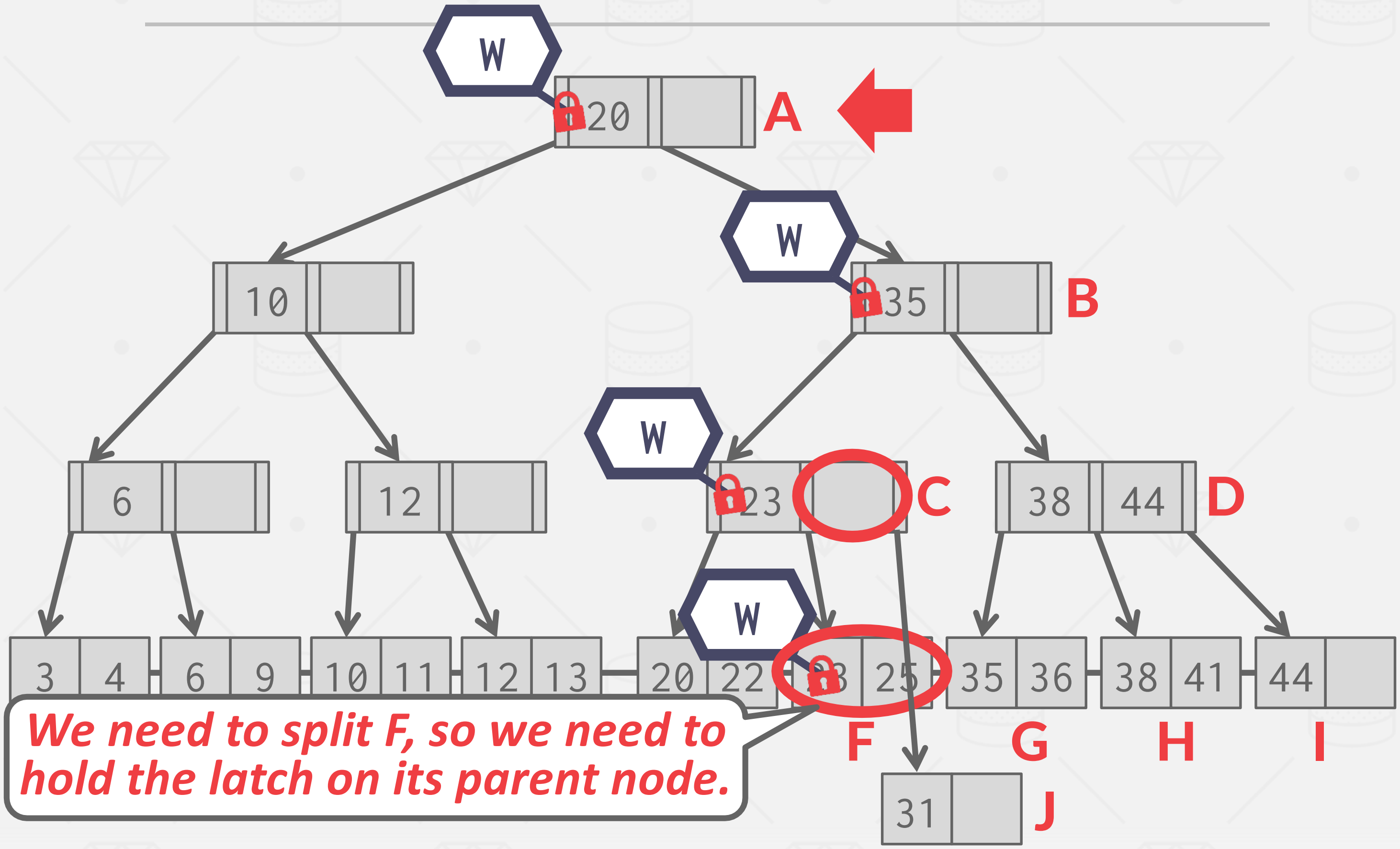
EXAMPLE #2 – DELETE 38



EXAMPLE #3 – INSERT 45

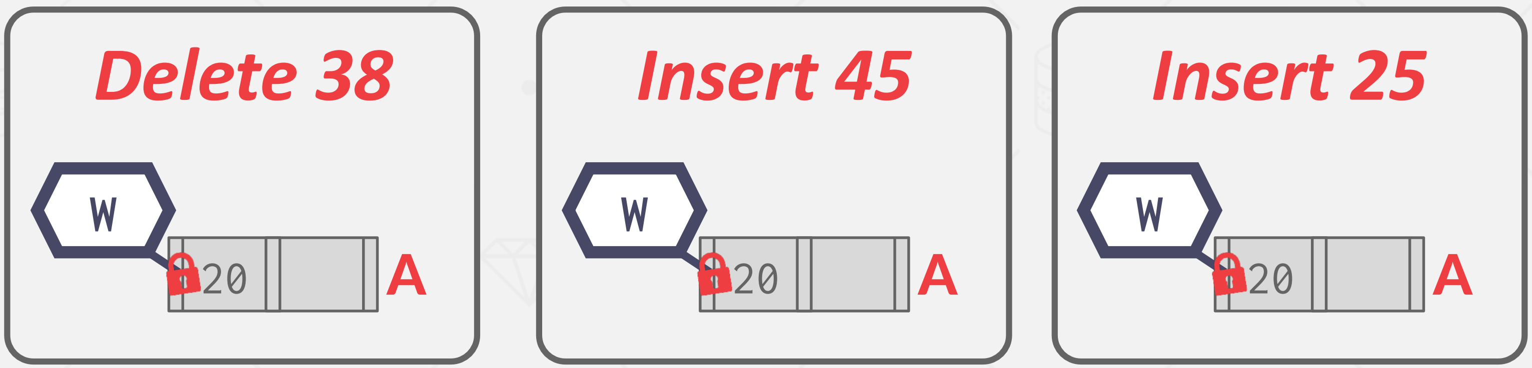


EXAMPLE #4 – INSERT 25



OBSERVATION

What was the first step that all the update examples did on the B+Tree?



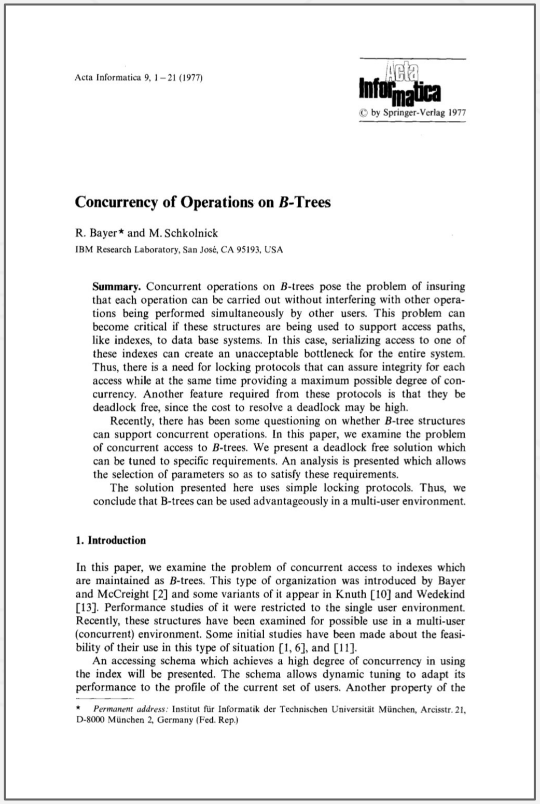
Taking a write latch on the root every time becomes a bottleneck with higher concurrency.

BETTER LATCHING ALGORITHM

Most modifications to a B+Tree will not require a split or merge.

Instead of assuming that there will be a split/merge, optimistically traverse the tree using read latches.

If you guess wrong, repeat traversal with the pessimistic algorithm.



Acta Informatica 9, 1 – 21 (1977)

Concurrency of Operations on *B*-Trees

R. Bayer* and M. Schkolnick
IBM Research Laboratory, San José, CA 95193, USA

Summary. Concurrent operations on *B*-trees pose the problem of insuring that each operation can be carried out without interfering with other operations being performed simultaneously by other users. This problem can become critical if these structures are being used to support access paths, like indexes, to data base systems. In this case, serializing access to one of these indexes can create an unacceptable bottleneck for the entire system. Thus, there is a need for locking protocols that can assure integrity for each access while at the same time providing a maximum possible degree of concurrency. Another feature required from these protocols is that they be deadlock free, since the cost to resolve a deadlock may be high.

Recently, there has been some questioning on whether *B*-tree structures can support concurrent operations. In this paper, we examine the problem of concurrent access to *B*-trees. We present a deadlock free solution which can be tuned to specific requirements. An analysis is presented which allows the selection of parameters so as to satisfy these requirements.

The solution presented here uses simple locking protocols. Thus, we conclude that B-trees can be used advantageously in a multi-user environment.

1. Introduction

In this paper, we examine the problem of concurrent access to indexes which are maintained as *B*-trees. This type of organization was introduced by Bayer and McCreight [2] and some variants of it appear in Knuth [10] and Wedekind [13]. Performance studies of it were restricted to the single user environment. Recently, these structures have been examined for possible use in a multi-user (concurrent) environment. Some initial studies have been made about the feasibility of their use in this type of situation [1, 6], and [11].

An accessing schema which achieves a high degree of concurrency in using the index will be presented. The schema allows dynamic tuning to adapt its performance to the profile of the current set of users. Another property of the

* *Permanent address:* Institut für Informatik der Technischen Universität München, Arcisstr. 21, D-8000 München 2, Germany (Fed. Rep.)

BETTER LATCHING ALGORITHM

Search: Same as before.

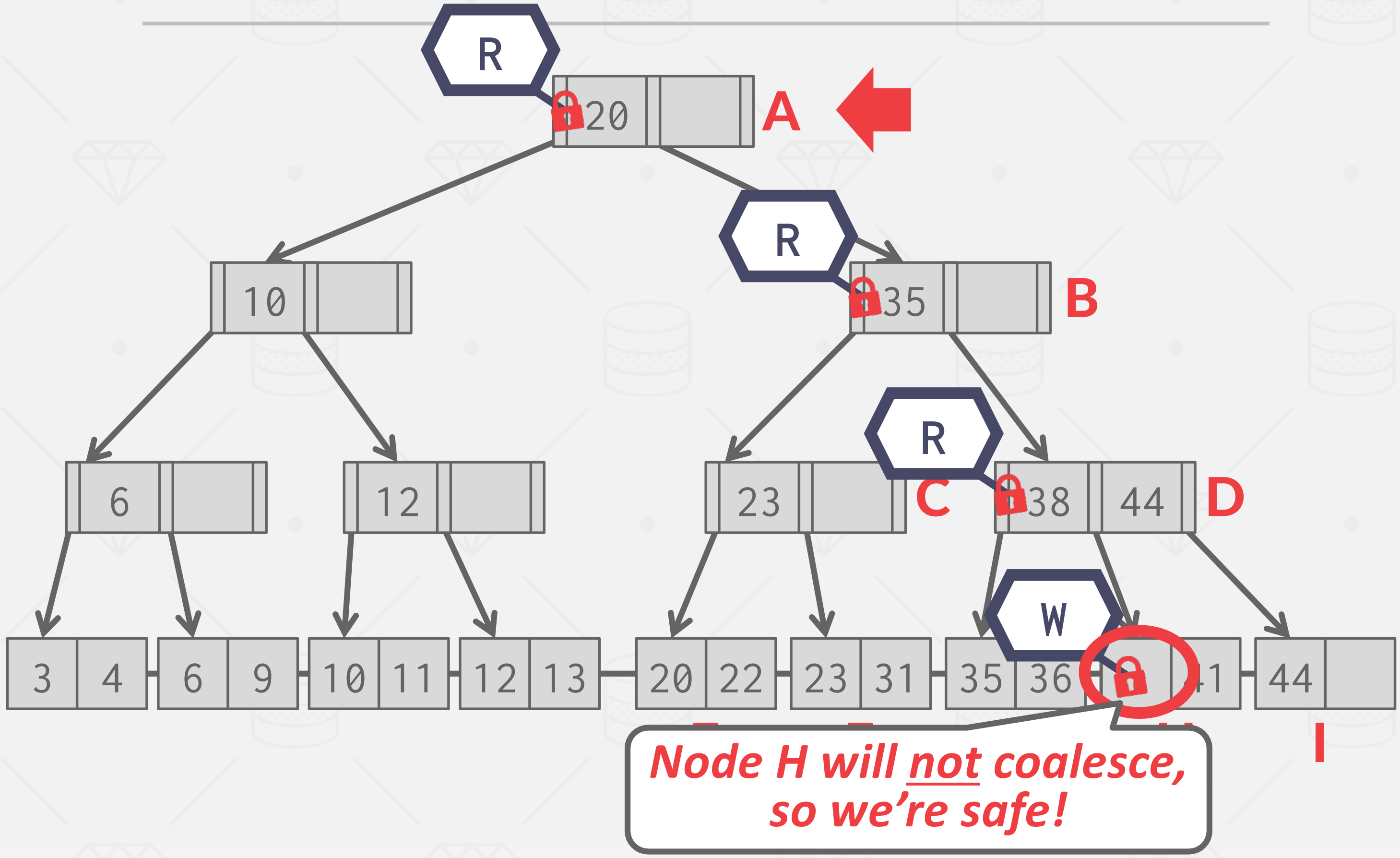
Insert/Delete:

→ Set latches as if for search, get to leaf, and set **W** latch on leaf.

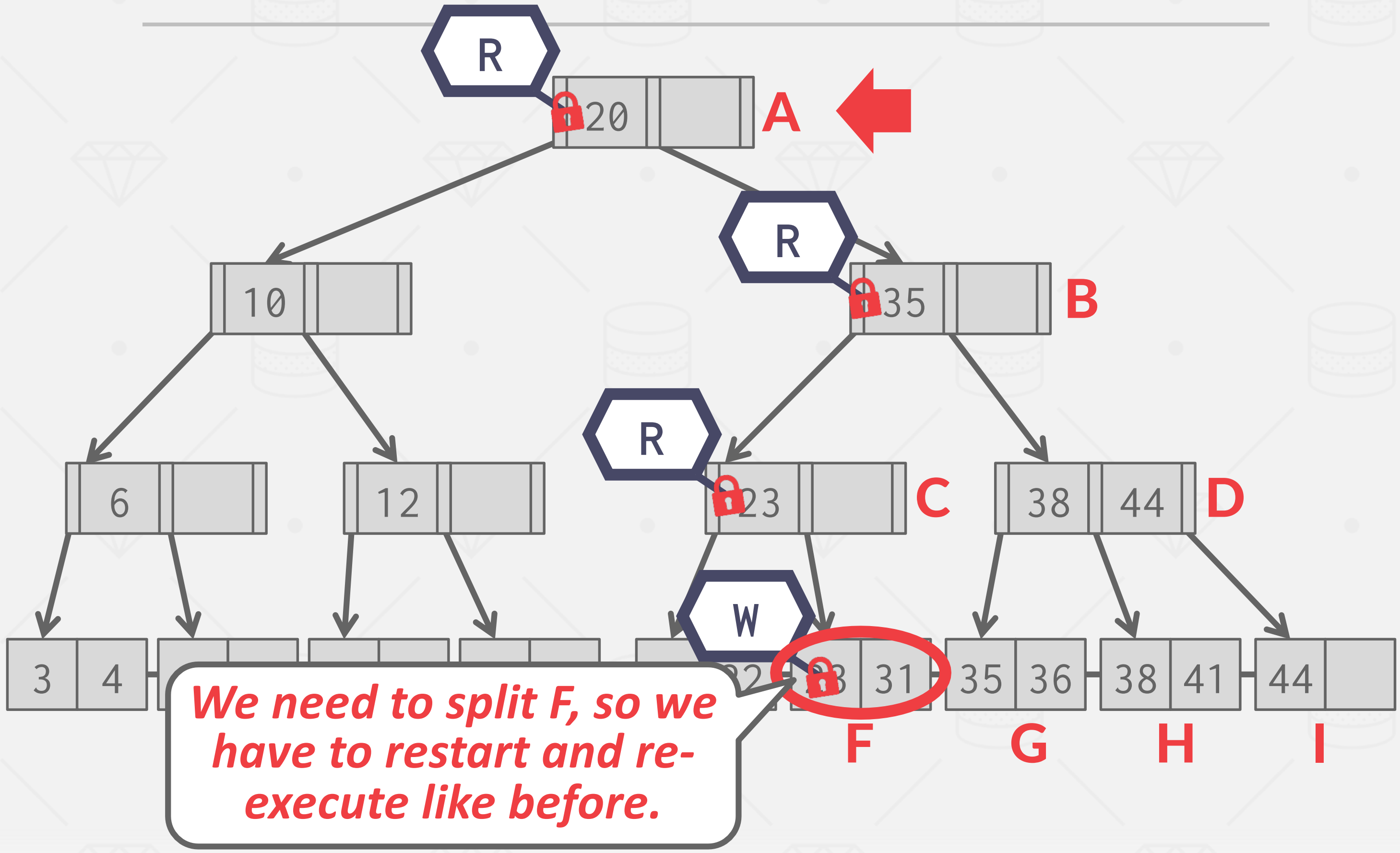
→ If leaf is not safe, release all latches, and restart thread using previous insert/delete protocol with write latches.

This approach optimistically assumes that only leaf node will be modified; if not, **R** latches set on the first pass to leaf are wasteful.

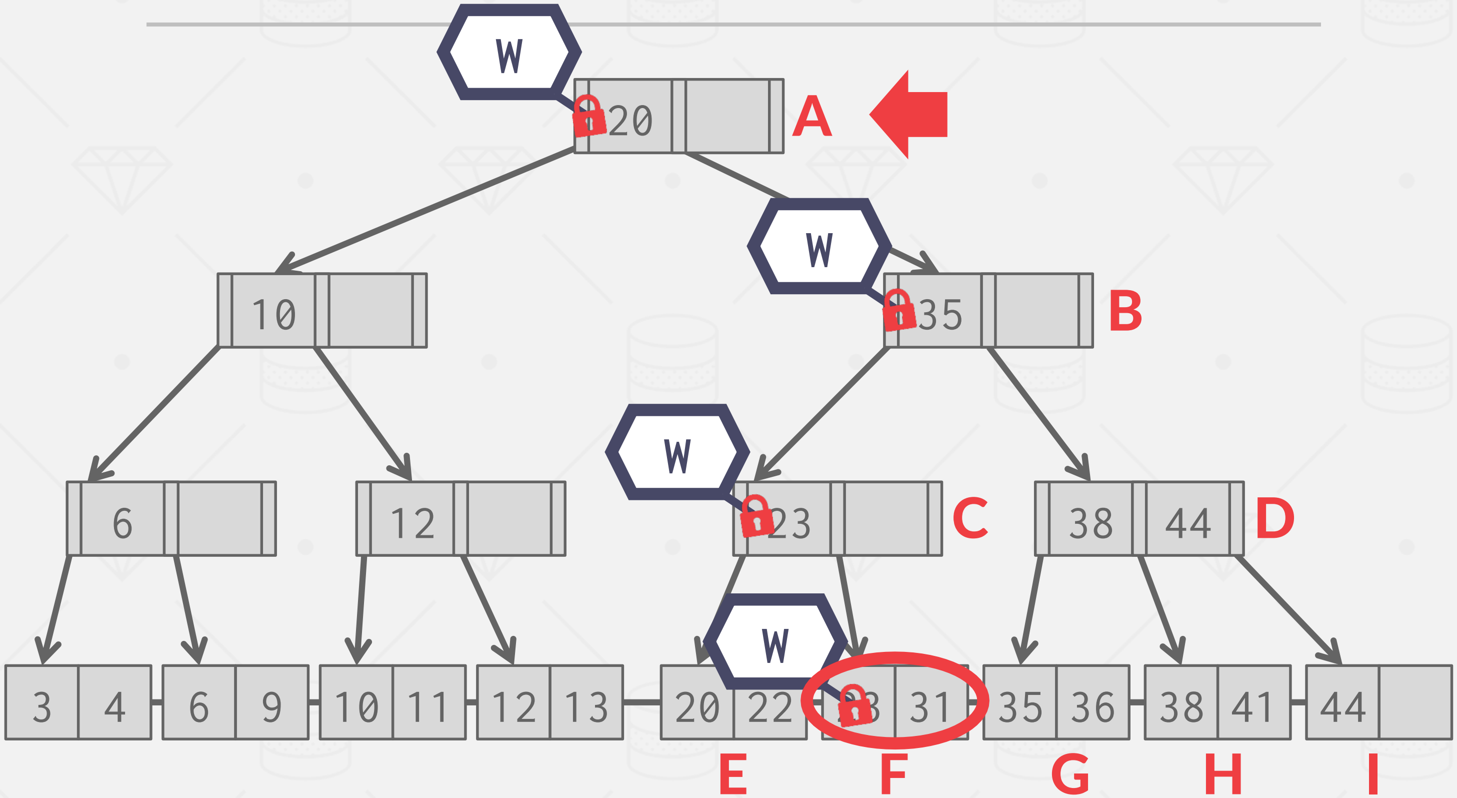
EXAMPLE #2 – DELETE 38



EXAMPLE #4 – INSERT 25



EXAMPLE #4 – INSERT 25



OBSERVATION

The threads in all the examples so far have acquired latches in a "top-down" manner.

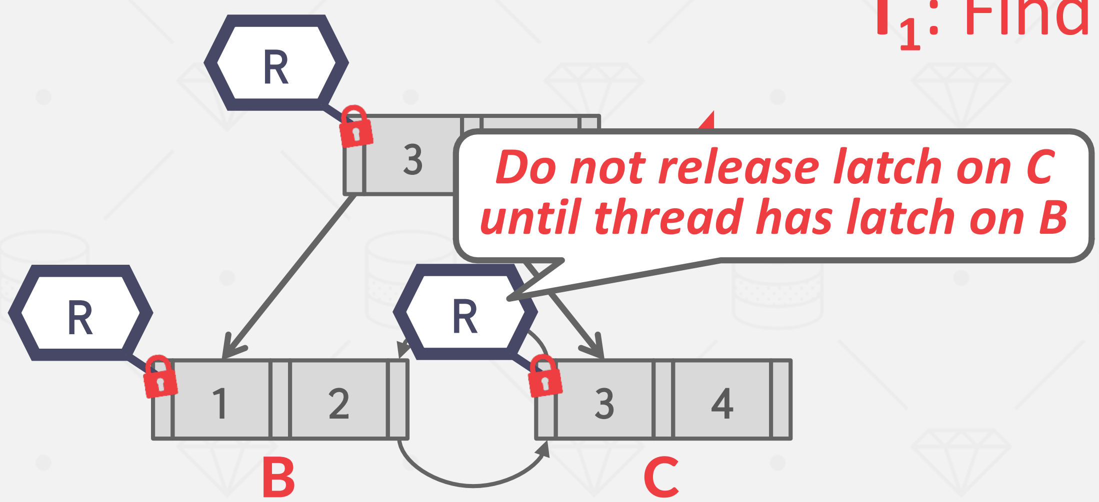
→ A thread can only acquire a latch from a node that is below its current node.

→ If the desired latch is unavailable, the thread must wait until it becomes available.

But what if threads want to move from one leaf node to another leaf node?

LEAF NODE SCAN EXAMPLE #1

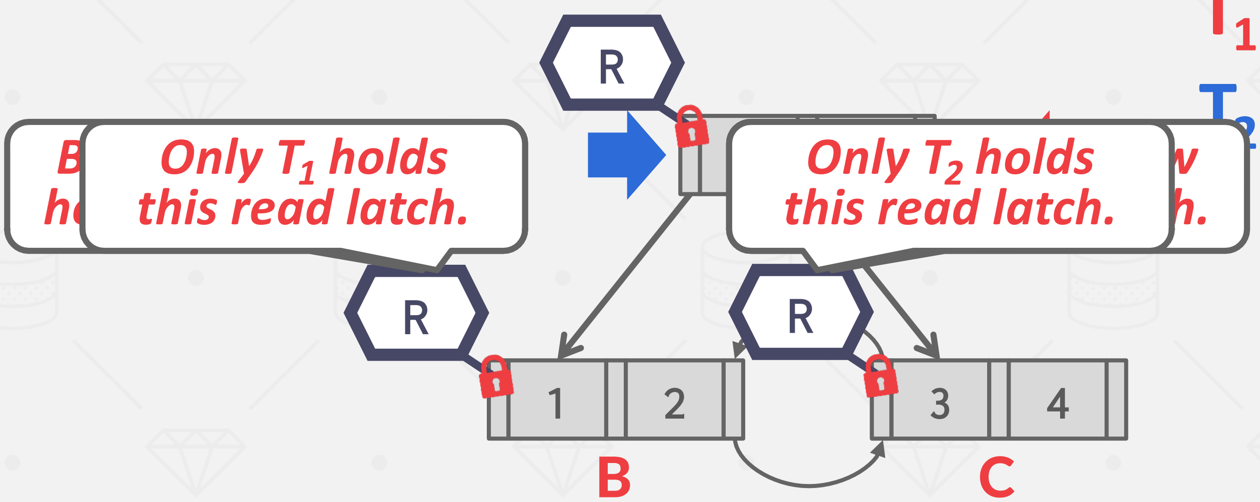
T$_1$: Find Keys < 4



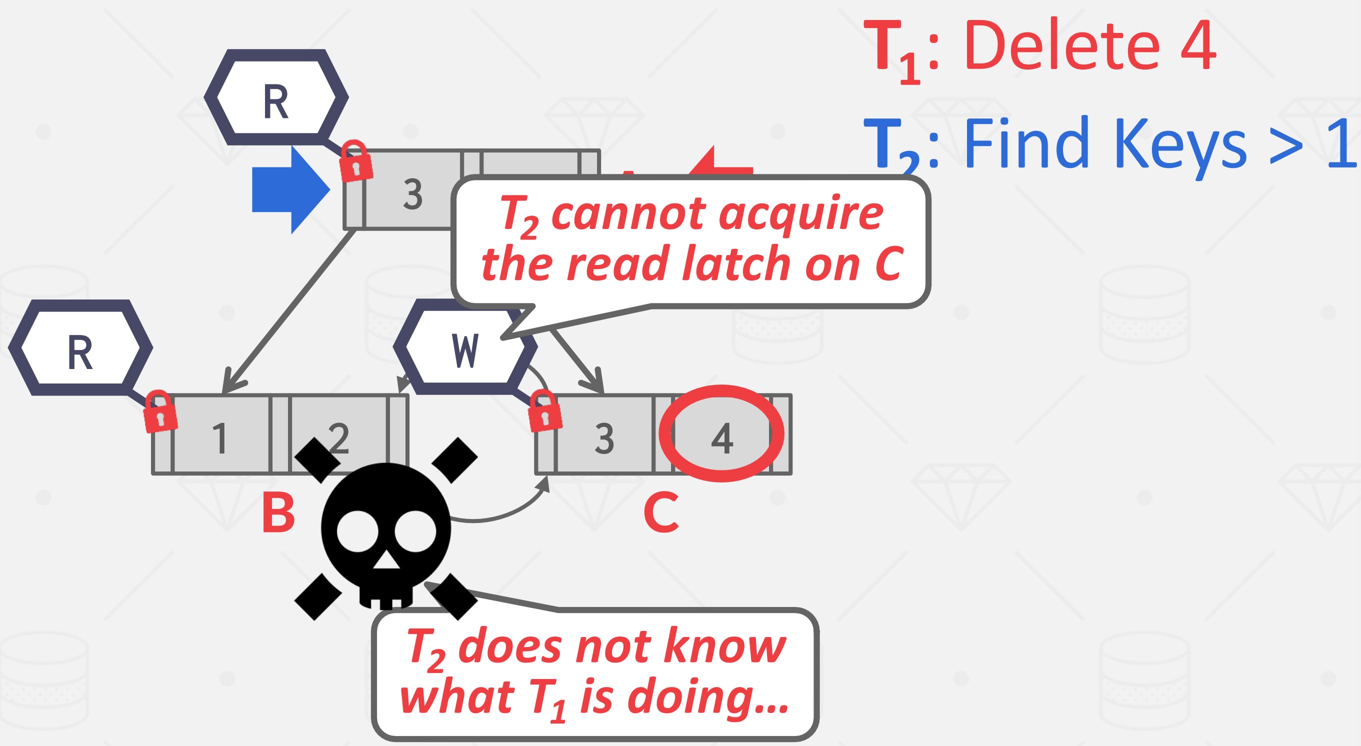
LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

T_2: Find Keys > 1



LEAF NODE SCAN EXAMPLE #3



LEAF NODE SCANS

Latches do not support deadlock detection or avoidance. The only way we can deal with this problem is through coding discipline.

The leaf node sibling latch acquisition protocol must support a "no-wait" mode.

The DBMS's data structures must cope with failed latch acquisitions.

CONCLUSION

Making a data structure thread-safe is notoriously difficult in practice.

We focused on B+Trees, but the same high-level techniques are applicable to other data structures.

NEXT CLASS

We are finally going to discuss how to execute some queries…