Lecture #09

Index

Concurrency Control
ADMINISTRIVIA

Project #1 is due Feb 18\textsuperscript{th} @ 11:59pm
→ Special Office Hours: Feb 17\textsuperscript{th} @ 3:00-5:00pm GHC 4405

Homework #2 is due Wed Feb 16\textsuperscript{th} @ 11:59pm

Mid-Term Exam is on Wednesday Feb 28\textsuperscript{th}
→ During regular class time from 12:30-1:50pm
→ Please contact us if you need accommodations.
→ Review session on Feb 26 during regular class time
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!
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 (Transactions)
→ Protect the database’s logical contents from other transactions.
→ Held for transaction's duration.
→ Need to be able to rollback changes.

Latches (Workers)
→ Protect the critical sections of the DBMS’s internal data structure from other workers (e.g., threads).
→ Held for operation duration.
→ Do not need to be able to rollback changes.
## Locks vs. Latches

<table>
<thead>
<tr>
<th>Locks</th>
<th>Latches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separate...</td>
<td>Transactions</td>
</tr>
<tr>
<td>Protect...</td>
<td>Database Contents</td>
</tr>
<tr>
<td>During...</td>
<td>Entire Transactions</td>
</tr>
<tr>
<td>Modes...</td>
<td>Shared, Exclusive, Update, Intention</td>
</tr>
<tr>
<td>Deadlock</td>
<td>Detection &amp; Resolution</td>
</tr>
<tr>
<td>...by...</td>
<td>Waits-for, Timeout, Aborts</td>
</tr>
<tr>
<td>Kept in...</td>
<td>Lock Manager</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Write</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
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 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 its own queue to track waiting threads.

By: Linus Torvalds (torvalds.delele@this.linux-foundation.org). January 3, 2020 6:05 pm

Beastial (no-mail.delele@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 again after acquiring the lock again, and claims that the time difference is the time when no lock was held. Which is just insane 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 be just after you read the "current time", but before you actually released the spinlock.

So now you still hold the lock, but you got scheduled away from the CPU, because you had used up your time slice. The "current time" you read is basically now stale, and has nothing to do with the (future) time when you are actually going to release the lock.

Somebody else comes in and wants that "spinlock", and that somebody will now spin for a long while, since nobody is releasing it - it's still held by that other thread entirely that was just scheduled out. At some point, the scheduler says "Ok, now you've used your time slice", and the thread and says "Oh, a long time passed without the lock being held at all".

And notice how the above is the good scenario. If you have more threads than CPUs (maybe because of other processes unrelated to your own test load), maybe the next thread that gets scheduled isn't the one that is going to release the lock. No, that one already got its running time now!

So the code in question is pure garbage. You can't do spinlocks like that. Or rather, you very much can do them like that, and when you do that you are measuring random latencies and getting nonsensical values, because what you are measuring is "I have a lot of busywork, where all the processes are CPU bound, and I'm measuring random points of how long the scheduler kept the process in place."

And then you write a blog-post blaming others, not understanding that it's your incorrect code that is garbage, and is giving random garbage values.
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 its own queue to track waiting threads.

---

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

Beastian (no.email.delete@this.xol.com) on January 3, 2020 11:40 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 I'm 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 again after acquiring the lock again, and claims that the time difference is the time when no lock was held. Which is just naive 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 or might not be additional

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.
LATCH IMPLEMENTATIONS

Test-and-Set Spinlock
Blocking OS Mutex
Reader-Writer Locks

Advanced approaches:
→ Adaptive Spinlock (Apple ParkingLot)
→ Queue-based Spinlock (MCS Locks)
LATCH IMPLEMENTATIONS

Approach #1: Test-and-Set Spin Latch (TAS)
→ Very efficient (single instruction to latch/unlatch)
→ Non-scalable, not cache friendly, not OS friendly.
→ Example: `std::atomic<T>`

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

Approach #2: Blocking OS Mutex

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

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

**Approach #2: 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: 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: Blocking OS Mutex

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

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

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

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

- OS Latch
- Userspace Latch
LATCH IMPLEMENTATIONS

Approach #3: 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`

Latch

![Latch Diagram]

- `read`:
  - `=0`
  - `=0`

- `write`:
  - `=0`
  - `=0`
LATCH IMPLEMENTATIONS

Approach #3: 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`
LATCH IMPLEMENTATIONS

Approach #3: 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

Latch

read
= 1
= 0

write
= 0
= 0
LATCH IMPLEMENTATIONS

Approach #3: 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`
LATCH IMPLEMENTATIONS

Approach #3: 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` \(\rightarrow\) `pthread_rwlock`

```
Latch
read = 2
write = 0
```
LATCH IMPLEMENTATIONS

Approach #3: 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`

**Latch**

- read = 2
- write = 0
- = 1
LATCH IMPLEMENTATIONS

Approach #3: 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

Latch

read

write

2

0

1
**LATCH IMPLEMENTATIONS**

**Approach #3: 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`

![Diagram showing reader-writer latch implementation](image)
COMPARE-AND-SWAP

Atomic instruction that compares contents of a memory location $M$ to a given value $V$

→ If values are equal, installs new given value $V'$ in $M$
→ Otherwise, operation fails

```c
__sync_bool_compare_and_swap(&M, 20, 30)
```
Atomic instruction that compares contents of a memory location $M$ to a given value $V$

→ If values are equal, installs new given value $V'$ in $M$

→ Otherwise, operation fails

```c
__sync_bool_compare_and_swap(&M, 20, 30)
```
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/Block-level Latches

→ Each page/block has its own reader-writer latch that protects its entire contents.
→ Threads acquire either a read or write latch before they access a page/block.

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/BLOCK LATCHES**

$T_1$: Find $D$

$hash(D)$
HASH TABLE – PAGE/BLOCK LATCHES

$T_1$: Find D

$hash(D)$
HASH TABLE - PAGE/BLOCK LATCHES

$T_1$: Find D  
$hash(D)$

$T_2$: Insert E  
$hash(E)$
**HASH TABLE – PAGE/BLOCK LATCHES**

\[ T_1: \text{Find D} \]

\[ \text{hash}(D) \]

\[ T_2: \text{Insert E} \]

\[ \text{hash}(E) \]
**HASH TABLE – PAGE/BLOCK LATCHES**

\[ T_1: \text{Find } D \]

\[ \text{hash}(D) \]

\[ T_2: \text{Insert } E \]

\[ \text{hash}(E) \]
**HASH TABLE - PAGE/BLOCK LATCHES**

T₁: Find D

hash(D)

T₂: Insert E

hash(E)
**HASH TABLE – PAGE/BLOCK LATCHES**

**T₁: Find D**

\(\text{hash}(D)\)

1. \(T_1: \text{Find } D\)
2. \(\text{hash}(E)\)
3. \(T_2: \text{Insert } E\)
**HASH TABLE – PAGE/BLOCK LATCHES**

- **$T_1$: Find $D$**
  - $\text{hash}(D)$

- **$T_2$: Insert $E$**
  - $\text{hash}(E)$

It’s safe to release the latch on Page #1.

- Page 0: $B | \text{val}$
- Page 1: $A | \text{val}$, $C | \text{val}$, $D | \text{val}$
- Page 2: Empty
**T₁: Find D**

\( \text{hash}(D) \)

**T₂: Insert E**

\( \text{hash}(E) \)
**HASH TABLE – PAGE/BLOCK LATCHES**

**T₁: Find D**

\[ \text{hash}(D) \]

**T₂: Insert E**

\[ \text{hash}(E) \]
**T₁: Find D**

hash(D)

**T₂: Insert E**

hash(E)

Hash Table - Page/Block Latches

- **B|val**
- **A|val**
- **D|val**
- **C|val**
**HASH TABLE - PAGE/BLOCK LATCHES**

**T₁: Find D**

\[ \text{hash}(D) \]

**T₂: Insert E**

\[ \text{hash}(E) \]
**HASH TABLE – PAGE/BLOCK LATCHES**

\[ T_1: \text{Find D} \]

\[ \text{hash(D)} \]

\[ T_2: \text{Insert E} \]

\[ \text{hash(E)} \]
HASH TABLE - PAGE/BLOCK LATCHES

$T_1$: Find D
$hash(D)$

$T_2$: Insert E
$hash(E)$
**HASH TABLE – PAGE/BLOCK LATCHES**

**T₁: Find D**

\[ \text{hash}(D) \]

**T₂: Insert E**

\[ \text{hash}(E) \]
**HASH TABLE – PAGE/BLOCK LATCHES**

**$T_1$: Find D**

$hash(D)$

**$T_2$: Insert E**

$hash(E)$

```
 hash(D)  
-------------------  
 A | val  
 C | val  
 D | val  
 E | val  
```

```
 W  
```
**HASH TABLE – SLOT LATCHES**

**T₁: Find D**

\[ \text{hash}(D) \]

**T₂: Insert E**

\[ \text{hash}(E) \]
**HASH TABLE – SLOT LATCHES**

**T₁: Find D**

\[ \text{hash}(D) \]

**T₂: Insert E**

\[ \text{hash}(E) \]
**HASH TABLE – SLOT LATCHES**

T₁: Find D  
hash(D)

T₂: Insert E  
hash(E)

---

R

<table>
<thead>
<tr>
<th>Slot</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>D</td>
</tr>
</tbody>
</table>

---
HASH TABLE - SLOT LATCHES

$T_1$: Find $D$

$hash(D)$

$T_2$: Insert $E$

$hash(E)$
**HASH TABLE – SLOT LATCHES**

**T₁: Find D**

\[ \text{hash}(D) \]

**T₂: Insert E**

\[ \text{hash}(E) \]
**HASH TABLE – SLOT LATCHES**

**T₁: Find D**

\(\text{hash}(D)\)

**T₂: Insert E**

\(\text{hash}(E)\)
$T_1$: Find $D$

$hash(D)$

It's safe to release the latch on $A$

$T_2$: Insert $E$

$hash(E)$
**HASH TABLE – SLOT LATCHES**

**$T_1$: Find D**

$\text{hash}(D)$

**$T_2$: Insert E**

$\text{hash}(E)$

- Find D: Hash D and search for its slot.
- Insert E: Hash E, check if the slot is free, and insert E if available.
**HASH TABLE – SLOT LATCHES**

**T₁: Find D**

hash(D)

**T₂: Insert E**

hash(E)
HASH TABLE - SLOT LATCHES

$T_1$: Find D

$hash(D)$

$T_2$: Insert E

$hash(E)$
$T_1$: Find $D$

$\text{hash}(D)$

$T_2$: Insert $E$

$\text{hash}(E)$
HASH TABLE – SLOT LATCHES

$T_1$: Find $D$

hash($D$)

$T_2$: Insert $E$

hash($E$)
**HASH TABLE – SLOT LATCHES**

\[ T_1: \text{Find } D \]
\[ \text{hash}(D) \]

\[ T_2: \text{Insert } E \]
\[ \text{hash}(E) \]
**HASH TABLE – SLOT LATCHES**

**T₁:** Find D  
$\text{hash}(D)$

**T₂:** Insert E  
$\text{hash}(E)$
**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

A

B

C

D

E

F

G

H

I

T1: Delete 44

T2: Find 41

3 4 6 9 10 11 12 13 20 22 23 31 35 36 38 41 44
B+TREE MULTI-THREADED EXAMPLE

T₁: Delete 44
B+TREE MULTI-THREADED EXAMPLE

T₁: Delete 44
**B+TREE MULTI-THREADED EXAMPLE**

\[ T_1: \text{Delete 44} \]
B+TREE MULTI-THREADED EXAMPLE

T₁: Delete 44
**B+TREE MULTI-THREADED EXAMPLE**

**T_1**: Delete 44

Rebalance!
**B+TREE MULTI-THREADED EXAMPLE**

**T₁:** Delete 44

Rebalance!
**B+Tree Multi-threaded Example**

**T₁: Delete 44**

- **A**: Root node with key 20.
- **B**: Node with key 35.
- **C**: Node with key 23.
- **D**: Node with key 44.
- **E**: Node with keys 3, 4, 6, 9.
- **F**: Node with keys 10, 11, 12, 13.
- **G**: Node with keys 35, 36, 38, 41.
- **H**: Node with key 20.
- **I**: Node with key 22.

Rebalance!
**B+TREE MULTI-THREADED EXAMPLE**

**T₁:** Delete 44  
**T₂:** Find 41  

Rebalance!
B+TREE MULTI-THREADED EXAMPLE

T₁: Delete 44
T₂: Find 41

Rebalance!
B+TREE MULTI-THREADED EXAMPLE

T₁: Delete 44
T₂: Find 41

Rebalance!
**B+TREE MULTI-THREADED EXAMPLE**

- **T₁:** Delete 44
- **T₂:** Find 41

Rebalance!
B+TREE MULTI-THREADED EXAMPLE

T₁: Delete 44
T₂: Find 41

Rebalance!
B+TREE MULTI-THREADED EXAMPLE

T₁: Delete 44
T₂: Find 41

Rebalance!
B+TREE MULTI-THREADED EXAMPLE

T₁: Delete 44
T₂: Find 41
**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 #1 – FIND 38

- Tree traversal example
  - Root: 20
  - Left child: 10
    - Left child: 6
      - Left child: 3
      - Right child: 4
    - Right child: 12
      - Left child: 10
      - Right child: 11
  - Right child: 35
    - Left child: 23
      - Left child: 20
      - Right child: 22
    - Right child: 38
      - Left child: 35
      - Right child: 36
      - Right child: 38
      - Right child: 41
      - Right child: 44
EXAMPLE #1 – FIND 38
EXAMPLE #1 – FIND 38

It is now safe to release the latch on A.
EXAMPLE #1 – FIND 38
EXAMPLE #1 – FIND 38

Diagram of a binary tree with nodes labeled from 3 to 44, showing the path to find the value 38.
EXAMPLE #1 – FIND 38

A

20

B

35

C

38

D

44

E

3

4

6

9

10

11

12

13

20

22

23

31

35

36

38

41

G

H

I

R
EXAMPLE #1 – FIND 38
EXAMPLE #1 – FIND 38
EXAMPLE #2 – DELETE 38
EXAMPLE #2 – DELETE 38
EXAMPLE #2 - DELETE 38
We may need to coalesce B, so we can’t release the latch on A.
EXAMPLE #2 – DELETE 38
We know that D will not merge with C, so it is safe to release latches on A and B.
EXAMPLE #2 – DELETE 38

We know that D will not merge with C, so it is safe to release latches on A and B.
EXAMPLE #2 – DELETE 38
EXAMPLE #2 – DELETE 38
EXAMPLE #2 – DELETE 38
EXAMPLE #3 – INSERT 45

A

B

C

D

E

F

G

H

I

20

35

10

23

6

22

12

31

9

11

12

36

38

44

3

4

6

9

10

11

20

23

38

44

45
EXAMPLE #3 – INSERT 45

Diagram of a binary search tree with nodes labeled from 3 to 44, illustrating the insertion of 45.
EXAMPLE #3 – INSERT 45

We know that if D needs to split, B has room so it is safe to release the latch on A.
EXAMPLE #3 – INSERT 45
EXAMPLE #3 – INSERT 45

Node I will not split, so we can release B+D.
**EXAMPLE #3 – INSERT 45**

Node I will not split, so we can release B+D.
EXAMPLE #3 – INSERT 45

Node I will not split, so we can release B+D.
EXAMPLE #3 – INSERT 45
EXAMPLE #4 – INSERT 25
EXAMPLE #4 – INSERT 25
EXAMPLE #4 – INSERT 25
EXAMPLE #4 – INSERT 25
EXAMPLE #4 – INSERT 25
EXAMPLE #4 – INSERT 25

Diagram showing a binary tree with nodes labeled 20, 10, 35, 6, 12, 3, 4, 6, 9, 10, 11, 12, 13, 20, 22, 13, 31, 23, 38, 44, 36, 38, 41, 44.
We need to split F, so we need to hold the latch on its parent node.
EXAMPLE #4 – INSERT 25

We need to split F, so we need to hold the latch on its parent node.
EXAMPLE #4 – INSERT 25

We need to split $F$, so we need to hold the latch on its parent node.
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.
**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 #2 – DELETE 38
EXAMPLE #2 – DELETE 38
EXAMPLE #2 - DELETE 38
EXAMPLE #2 – DELETE 38
EXAMPLE #2 – DELETE 38
EXAMPLE #2 – DELETE 38

Node H will not coalesce, so we’re safe!
EXAMPLE #2 – DELETE 38

Node H will not coalesce, so we’re safe!
EXAMPLE #2 – DELETE 38

Node H will not coalesce, so we’re safe!
EXAMPLE #4 – INSERT 25
We need to split F, so we have to restart and re-execute like before.
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

A

B

C

1 2

3 4

131
LEAF NODE SCAN EXAMPLE #1

T₁: Find Keys < 4
LEAF NODE SCAN EXAMPLE #1

$T_1$: Find Keys $< 4$
LEAF NODE SCAN EXAMPLE #1

$T_1$: Find Keys < 4
LEAF NODE SCAN EXAMPLE #1

$T_1$: Find Keys < 4

Do not release latch on C until thread has latch on B
LEAF NODE SCAN EXAMPLE #1

$T_1$: Find Keys < 4

Do not release latch on C until thread has latch on B
LEAF NODE SCAN EXAMPLE #1

$T_1$: Find Keys < 4
LEAF NODE SCAN EXAMPLE #2

\[ T_1: \text{Find Keys } < 4 \]
\[ T_2: \text{Find Keys } > 1 \]
LEAF NODE SCAN EXAMPLE #2

$T_1$: Find Keys < 4

$T_2$: Find Keys > 1
LEAF NODE SCAN EXAMPLE #2

$T_1$: Find Keys < 4

$T_2$: Find Keys > 1
LEAF NODE SCAN EXAMPLE #2

\[ T_1: \text{Find Keys } < 4 \]
\[ T_2: \text{Find Keys } > 1 \]
LEAF NODE SCAN EXAMPLE #2

\[ T_1: \text{Find Keys < 4} \]
\[ T_2: \text{Find Keys > 1} \]
LEAF NODE SCAN EXAMPLE #2

$T_1$: Find Keys < 4

$T_2$: Find Keys > 1
LEAF NODE SCAN EXAMPLE #2

Both $T_1$ and $T_2$ now hold this read latch.

$T_1$: Find Keys < 4

$T_2$: Find Keys > 1

Both $T_1$ and $T_2$ now hold this read latch.
LEAF NODE SCAN EXAMPLE #2

Both $T_1$ and $T_2$ now hold this read latch.

$T_1$: Find Keys < 4
$T_2$: Find Keys > 1

Both $T_1$ and $T_2$ now hold this read latch.
LEAF NODE SCAN EXAMPLE #2

$T_1$: Find Keys < 4

$T_2$: Find Keys > 1

Only $T_1$ holds this read latch.

Only $T_2$ holds this read latch.
LEAF NODE SCAN EXAMPLE #3

A

B

C

T1: Delete 4

T2: Find Keys > 1

Choices?
LEAF NODE SCAN EXAMPLE #3

$T_1$: Delete 4
$T_2$: Find Keys > 1
LEAF NODE SCAN EXAMPLE #3

\( T_1: \) Delete 4

\( T_2: \) Find Keys > 1

\[ \begin{array}{c}
1 & 2 & \leftarrow A \rightarrow 3 \\
3 & 4 & \leftarrow B \rightarrow C
\end{array} \]
LEAF NODE SCAN EXAMPLE #3

\[ T_1: \text{Delete 4} \]
\[ T_2: \text{Find Keys > 1} \]
LEAF NODE SCAN EXAMPLE #3

$T_1$: Delete 4

$T_2$: Find Keys $> 1$
LEAF NODE SCAN EXAMPLE #3

$T_1$: Delete 4

$T_2$: Find Keys > 1

$T_2$ cannot acquire the read latch on C
LEAF NODE SCAN EXAMPLE #3

$T_1$: Delete 4  
$T_2$: Find Keys > 1

$T_2$ cannot acquire the read latch on C

$T_2$ does not know what $T_1$ is doing...
**LEAF NODE SCAN EXAMPLE #3**

**T₂ Choices?**

- **T₁**: Delete 4
- **T₂**: Find Keys > 1

- **T₂** cannot acquire the read latch on C

- **T₂** does not know what **T₁** is doing...
LEAF NODE SCAN EXAMPLE #3

T₂ Choices?

Wait

T₁: Delete 4
T₂: Find Keys > 1

T₂ cannot acquire the read latch on C

T₂ does not know what T₁ is doing...
LEAF NODE SCAN EXAMPLE #3

**$T_1$: Delete 4**

**$T_2$: Find Keys > 1**

**$T_2$ Choices?**

- **Wait**
- **Kill Ourself**

$T_2$ cannot acquire the read latch on C

$T_2$ does not know what $T_1$ is doing...
LEAF NODE SCAN EXAMPLE #3

**T₁**: Delete 4

**T₂**: Find Keys > 1

**T₂** Choices?
- Wait
- Kill Ourselves
- Kill Other Thread

*T₂* does not know what *T₁* is doing...
*T₂* cannot acquire the read latch on C
**LEAF NODE SCAN EXAMPLE #3**

**T₂ Choices?**
- Wait
- Kill Ourselves
- Kill Other Thread

**T₁:** Delete 4
**T₂:** Find Keys > 1

*T₂* cannot acquire the read latch on C

*T₂* does not know what *T₁* is doing...
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.
We are finally going to discuss how to execute some queries...