Index Concurrency Control
**ADMINISTRIVIA**

**Project #1** was due last night @ 11:59pm

**Homework #2** is due Sunday, Oct 3rd @ 11:59pm

**Project #2** will be released today and is due on Sunday, Oct 17th @ 11:59pm
QUESTIONS FROM LAST CLASS

(1) Non-prefix lookups in multi-attribute B+Trees
(2) Efficiently merging B+Trees
The DBMS can use a B+Tree index if the query provides any of the attributes of the search key.

Example: Index on \(<a, b, c>\)
→ Supported: \((a=5 \text{ AND } b=3)\)
→ Supported: \((b=3)\)

Not all DBMSs support this.
For a hash index, we must have all attributes in search key.
SELECTION CONDITIONS

Find Key=(A,B)
Find Key=(A,*)
Find Key=(A,B)

Find Key=(A,*)
Example: Index on \(<col1, col2, col3>\)
→ Column Values: \{A, B, C, D\}
→ Supported: \(col2 = B\)
**Example:** Index on $<\text{col1}, \text{col2}, \text{col3}>$

$\rightarrow$ Column Values: $\{A, B, C, D\}$

$\rightarrow$ Supported: $\text{col2} = B$
Example: Index on $<col1, col2, col3>$

→ Column Values: \{A,B,C,D\}

→ Supported: col2 = B
**Example:** Index on \( <\text{col1}, \text{col2}, \text{col3}> \)

→ Column Values: \{A, B, C, D\}

→ Supported: \( \text{col2} = B \)

---

**SELECTION CONDITIONS**

---

Example: Index on \( <\text{col1}, \text{col2}, \text{col3}> \)

→ Column Values: \{A, B, C, D\}

→ Supported: \( \text{col2} = B \)
Example: Index on <col1, col2, col3>
→ Column Values: \{A, B, C, D\}
→ Supported: \textit{col2} = B
**Example:** Index on \(<\text{col1}, \text{col2}, \text{col3}>\)

→ Column Values: \{A, B, C, D\}
→ Supported: \text{col2} = B
**Example:** Index on \(<col1, col2, col3>\)

→ Column Values: \{A, B, C, D\}

→ Supported: \(col2 = B\)
**Example:** Index on `<col1, col2, col3>`

→ Column Values: `{A, B, C, D}`

→ Supported: `col2 = B`
**Example:** Index on \(<\text{col1}, \text{col2}, \text{col3}>\) 
→ Column Values: \{\text{A, B, C, D}\} 
→ Supported: \text{col2} = \text{B}
**Example:** Index on \(<\text{col1}, \text{col2}, \text{col3}>\)  
→ Column Values: \{A, B, C, D\}  
→ Supported: \text{col2} = B
Example: Index on <col1, col2, col3>
→ Column Values: \{A, B, C, D\}
→ Supported: col2 = B
Example: Index on \(<\text{col1, col2, col3}>\)

→ Column Values: \{A, B, C, D\}

→ Supported: \text{col2} = B
Example: Index on <col1, col2, col3>
→ Column Values: {A, B, C, D}
→ Supported: col2 = B
Example: Index on \(<\text{col1}, \text{col2}, \text{col3}>\)

→ Column Values: \{A, B, C, D\}

→ Supported: \text{col2} = B
**Example**: Index on `<col1, col2, col3>`

→ Column Values: `{A, B, C, D}`

→ Supported: `col2 = B`
Example: Index on `<col1, col2, col3>`
→ Column Values: `{A, B, C, D}`
→ Supported: `col2 = B`
Example: Index on `<col1, col2, col3>`
→ Column Values: `{A, B, C, D}`
→ Supported: `col2 = B`
Example: Index on \(<\text{col1}, \text{col2}, \text{col3}>\)

→ Column Values: \{A, B, C, D\}

→ Supported: \text{col2} = B

“Skip Scan”
Merging B+Trees

Approach #1: Off-line
→ Block all operations until done merging.

Approach #2: Eager
→ Access both during merge; move batches eagerly.

Approach #3: Background
→ Copy + merge in background; apply missed updates.

Approach #4: Lazy
→ Designate one as main and other as secondary.
   → If leaf in main not yet updated, merge corresponding key range from secondary.
We (mostly) assumed all the data structures that we have discussed so far are single-threaded.

But we need to allow multiple threads to safely access our data structures to take advantage of additional CPU cores and hide disk I/O stalls.
We (mostly) assumed all the data structures that we have discussed so far are single-threaded.

But we need to allow multiple threads to safely access our data structures to take advantage of additional CPU cores and hide disk I/O stalls.
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?
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.
<table>
<thead>
<tr>
<th>Locks</th>
<th>Latches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separate...</td>
<td>User 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,</td>
</tr>
<tr>
<td></td>
<td>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](https://www.cmu-db.org)
# Locks vs. Latches

<table>
<thead>
<tr>
<th><strong>Locks</strong></th>
<th><strong>Latches</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Separate...</td>
<td>Threads</td>
</tr>
<tr>
<td>User Transactions</td>
<td>In-Memory Data Structures</td>
</tr>
<tr>
<td>Protect...</td>
<td>Critical Sections</td>
</tr>
<tr>
<td>Database Contents</td>
<td>Read, Write</td>
</tr>
<tr>
<td>During...</td>
<td>Avoidance</td>
</tr>
<tr>
<td>Entire Transactions</td>
<td>Coding Discipline</td>
</tr>
<tr>
<td>Modes...</td>
<td>Protected Data Structure</td>
</tr>
<tr>
<td>Shared, Exclusive, Update,</td>
<td></td>
</tr>
<tr>
<td>Intention</td>
<td></td>
</tr>
<tr>
<td>Deadlock</td>
<td></td>
</tr>
<tr>
<td>Detection &amp; Resolution</td>
<td></td>
</tr>
<tr>
<td>...by...</td>
<td></td>
</tr>
<tr>
<td>Waits-for, Timeout, Aborts</td>
<td></td>
</tr>
<tr>
<td>Kept in...</td>
<td></td>
</tr>
<tr>
<td>Lock Manager</td>
<td></td>
</tr>
</tbody>
</table>

Source: Goetz Graefe
# Locks vs. Latches

<table>
<thead>
<tr>
<th>Locks</th>
<th>Latches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separate...</td>
<td>Threads</td>
</tr>
<tr>
<td>Protect...</td>
<td>In-Memory Data Structures</td>
</tr>
<tr>
<td>During...</td>
<td>Critical Sections</td>
</tr>
<tr>
<td>Modes...</td>
<td>Read, Write</td>
</tr>
<tr>
<td>Deadlock</td>
<td>Avoidance</td>
</tr>
<tr>
<td>...by...</td>
<td>Coding Discipline</td>
</tr>
<tr>
<td>Kept in...</td>
<td>Protected Data Structure</td>
</tr>
</tbody>
</table>

- **Locks**
  - User Transactions
  - Database Contents
  - Entire Transactions
  - Shared, Exclusive, Update, Intention

- **Latches**
  - Waits-for, Timeout, Aborts
  - Lock Manager

Source: [Goetz Graefe](https://www.cs.cmu.edu/~cmu-db/)
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.
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>✗</td>
</tr>
<tr>
<td>Write</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
LATCH IMPLEMENTATIONS

Blocking OS Mutex
Test-and-Set Spin Latch
Reader-Writer Latches
LATCH IMPLEMENTATIONS

Approach #1: Blocking OS Mutex

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

Approach #1: 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();
```
Approach #1: 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

Approach #1: 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

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

```
std::mutex m;
pthread_mutex_t futex;

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

m.unlock();
```
LATCH IMPLEMENTATIONS

Approach #1: 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();
```
Approach #1: 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();
```

OS Latch

Userspace Latch
LATCH IMPLEMENTATIONS

Approach #1: 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();
```

OS Latch
Userspace Latch
Approach #1: Blocking OS Mutex

→ Simple to use
→ Non-scalable (about 25ns per lock/unlock invocation)
→ Example: `std::mutex`
Approach #2: 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>`
Approach #2: 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: 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: 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: 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?
}
```
Approach #2: Test

→ Very efficient (single instruction to latch/unlatch)
→ Non-scalable, not cache-friendly, not OS-friendly

→ Example:

```cpp
std::atomic<T> latch;

while (latch.test_and_set(...)) {
  // Retry? Yield? Abort?
}
```

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 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 schedules the original thread, and now the lock is actually released. Then another thread comes in, gets the lock again, and then it looks at the time 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 CPU’s (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 right 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.
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

Choice #3: Reader-Writer Latches

→ Allows for concurrent readers
→ Must manage read/write queues to avoid starvation
→ Can be implemented on top of spin latches
Choice #3: Reader-Writer Latches

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

Latch

- read: Latch = 0
- write: Latch = 0
LATCH IMPLEMENTATIONS

Choice #3: Reader-Writer Latches

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

Choice #3: Reader-Writer Latches

→ Allows for concurrent readers
→ Must manage read/write queues to avoid starvation
→ Can be implemented on top of spin latches
Choice #3: Reader-Writer Latches

→ Allows for concurrent readers
→ Must manage read/write queues to avoid starvation
→ Can be implemented on top of spin latches
Choice #3: Reader-Writer Latches

→ Allows for concurrent readers
→ Must manage read/write queues to avoid starvation
→ Can be implemented on top of spin latches
Choice #3: Reader-Writer Latches

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

Choice #3: Reader-Writer Latches
→ Allows for concurrent readers
→ Must manage read/write queues to avoid starvation
→ Can be implemented on top of spin latches
LATCH IMPLEMENTATIONS

Choice #3: Reader-Writer Latches

→ Allows for concurrent readers
→ Must manage read/write queues to avoid starvation
→ Can be implemented on top of spin latches
Choice #3: Reader-Writer Latches

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

Choice #3: Reader-Writer Latches
→ Allows for concurrent readers
→ Must manage read/write queues to avoid starvation
→ Can be implemented on top of spin latches

Latch

read

write

\[\begin{align*}
\text{read} & = 2 \\
\text{write} & = 0 \\
\text{read} & = 1 \\
\text{write} & = 1
\end{align*}\]
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

<table>
<thead>
<tr>
<th>A</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>val</td>
</tr>
<tr>
<td>C</td>
<td>val</td>
</tr>
<tr>
<td>D</td>
<td>val</td>
</tr>
</tbody>
</table>

15-445/645 (Fall 2021)
**HASH TABLE – PAGE LATCHES**

**T₁: Find D**

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

<table>
<thead>
<tr>
<th>B</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>val</td>
</tr>
<tr>
<td>C</td>
<td>val</td>
</tr>
<tr>
<td>D</td>
<td>val</td>
</tr>
</tbody>
</table>
$T_1$: Find D

$\text{hash}(D)$
**T₁: Find D**

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

**HASH TABLE – PAGE LATCHES**

- **B | val**
- **A | val**
- **C | val**
- **D | val**
$T_1$: Find $D$

$\text{hash}(D)$
Hash Table – Page Latches

$T_1$: Find D

hash(D)

$T_2$: Insert E

hash(E)

- B: val
- A: val
- C: val
- D: val
\( T_1: \) Find D

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

\( T_2: \) Insert E

\[ \text{hash}(E) \]
**T₁**: Find D

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

**T₂**: Insert E

\[\text{hash}(E)\]
T₁: Find D

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

T₂: Insert E

\( \text{hash}(E) \)
\textbf{HASH TABLE – PAGE LATCHES}

\textbf{T}_1: Find D \quad hash(D)

\textbf{T}_2: Insert E \quad hash(E)
**HASH TABLE – PAGE LATCHES**

**T₁: Find D**
\[ \text{hash}(D) \]

**T₂: Insert E**
\[ \text{hash}(E) \]

- **0**: B | val
- **1**: A | val
- **2**: C | val
- **3**: D | val
It’s safe to release the latch on Page #1.

$T_1$: Find $D$

$hash(D)$

$T_2$: Insert $E$

$hash(E)$

It's safe to release the latch on Page #1.
**HASH TABLE – PAGE LATCHES**

**T₁: Find D**
\[\text{hash}(D)\]

**T₂: Insert E**
\[\text{hash}(E)\]
**T₁: Find D**

**hash(D)**

**T₂: Insert E**

**hash(E)**
**T₁: Find D**
\[ \text{hash}(D) \]

**T₂: Insert E**
\[ \text{hash}(E) \]
**HASH TABLE – PAGE LATCHES**

**T₁: Find D**

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

**T₂: Insert E**

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

**T₁: Find D**

hash(D)

**T₂: Insert E**

hash(E)
## Hash Table – Page Latches

**T₁: Find D**

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

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>val</td>
</tr>
</tbody>
</table>

**T₂: Insert E**

\( \text{hash}(E) \)

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>val</td>
</tr>
<tr>
<td>C</td>
<td>val</td>
</tr>
<tr>
<td>D</td>
<td>val</td>
</tr>
<tr>
<td>E</td>
<td>val</td>
</tr>
</tbody>
</table>
**T₁: Find D**  
*hash(D)*

**T₂: Insert E**  
*hash(E)*
**HASH TABLE – SLOT LATCHES**

**T₁:** Find D  
\( \text{hash}(D) \)

**T₂:** Insert E  
\( \text{hash}(E) \)
**HASH TABLE – SLOT LATCHES**

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

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

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

\[ \text{hash}(E) \]
T₁: Find D

T₂: Insert E

hash(D)

hash(E)
**HASH TABLE – SLOT LATCHES**

**T₁: Find D**

$\text{hash}(D)$

**T₂: Insert E**

$\text{hash}(E)$
**Hash Table – Slot Latches**

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

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

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

\( \text{hash}(E) \)

<table>
<thead>
<tr>
<th>Slot</th>
<th>Value</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>D</td>
</tr>
</tbody>
</table>

\( R \)

\( W \)

\( A \)

\( C \)

\( D \)
**HASH TABLE — SLOT LATCHES**

**T_1:** Find D

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

**T_2:** Insert E

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

- **R:** Read
- **W:** Write

<table>
<thead>
<tr>
<th>Slot</th>
<th>Value</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>D</td>
</tr>
</tbody>
</table>
**HASH TABLE – SLOT LATCHES**

**T₁:**
- **hash(D)**
- **It’s safe to release the latch on A**

**T₂:** **Insert E**
- **hash(E)**

Events:
1. **T₁:** Find D
2. **T₂:** Insert E

Conditions:
- **B|val**: Occupied by B
- **C|val**: Occupied by C
- **D|val**: Occupied by D
- **A**: Latched by A
- **R**: Read
- **W**: Write
**T_1:** Find D  
\( \text{hash}(D) \)

**T_2:** Insert E  
\( \text{hash}(E) \)
$T_1$: Find $D$

$hash(D)$

$T_2$: Insert $E$

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

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

\[ T_2: \text{Insert } E \]
\[ \text{hash}(E) \]
$T_1$: Find $D$  
$hash(D)$

$T_2$: Insert $E$  
$hash(E)$
**T₁: Find D**

`hash(D)`

**T₂: Insert E**

`hash(E)`
**T₁:** Find D

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

**T₂:** Insert E

\( \text{hash}(E) \)
**T₁: Find D**

(hash(D))

**T₂: Insert E**

(hash(E))
HASH TABLE – NO LATCHES?

LINEAR PROBE HASHING

hash(key)

A
B
C
D
E
F

B | val
A | val
C | val
t
D | val
E | val
F | val
HASH TABLE – NO LATCHES?

LINEAR PROBE HASHING

hash(key)

| A | val |
| B | val |
| C | val |
| D | val |
| E | val |
| F | val |

22m48s
Atomic instruction that compares contents of a memory location $M$ to a given value $V$

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

$_{\text{sync\_bool\_compare\_and\_swap}}$(&$M$, 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

\[
\text{__sync_bool_compare_and_swap}(&M, 20, 30)
\]
Atomic instruction that compares contents of a memory location $M$ to a given value $V$

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

$\rightarrow$ Otherwise, operation fails

\_\_sync\_bool\_compare\_and\_swap$(\&M, 20, 30)$
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

Diagram of a B+ Tree with nodes labeled as follows:
- Node A: 20
- Node B: 35
- Node C: 23
- Node D: 38
- Node E: 3
- Node F: 10
- Node G: 20
- Node H: 20
- Node I: 44
**B+Tree Multi-Threaded Example**

$T_1$: Delete 44
B+TREE MULTI-THREADED EXAMPLE

T \_1: Delete 44
**B+TREE MULTI-THREADED EXAMPLE**

**T₁: Delete 44**

Diagram showing a B+ tree with nodes labeled A, B, C, D, E, F, G, H, I. Node B is highlighted with a delete marker indicating the removal of the value 44.
**B+TREE MULTI-THREADED EXAMPLE**

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

T₁: Delete 44

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

**T1:** Delete 44

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

T₁: Delete 44

Rebalance!
B+Tree Multi-Threaded Example

T₁: Delete 44
T₂: Find 41

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

1. **Delete 44**
   - Node A:Delete 44
   - Node B: Delete 44
   - Node C: Delete 44
   - Node D: Delete 44
   - Node E: Delete 44
   - Node F: Delete 44
   - Node G: Delete 44
   - Node H: Delete 44
   - Node I: Delete 44

2. **Find 41**
   - Node B: Find 41
   - Node C: Find 41
   - Node D: Find 41
   - Node E: Find 41
   - Node F: Find 41
   - Node G: Find 41
   - Node H: Find 41
   - Node I: Find 41

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

T₁: Delete 44
T₂: Find 41
**B+TREE MULTI-THREADED EXAMPLE**

- **T₁:** Delete 44
- **T₂:** Find 41
**B+Tree Multi-Threaded Example**

- **T₁:** Delete 44
- **T₂:** Find 41

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

\[ T_1: \text{Delete 44} \]
\[ T_2: \text{Find 41} \]
B+TREE MULTI-THREADED EXAMPLE

T₁: Delete 44
T₂: Find 41
Protocol to allow multiple threads to access/modify B+Tree at the same time.

**Basic Idea:**
- 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 go down; repeatedly,
→ Acquire \( R \) latch on child
→ Then unlatch parent

**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

A

10

B

35

C

23

D

38

E

3

F

6

G

10

H

12

I

9
EXAMPLE #1 – FIND 38

1. Start at the root node R.
2. Compare 38 with the node value 20. Since 38 > 20, move to the right child node.
3. Compare 38 with the node value 35. Since 38 < 35, move to the left child node.
4. Compare 38 with the node value 23. Since 38 > 23, move to the right child node.
5. Compare 38 with the node value 38. Since 38 = 38, the node with value 38 is found.

The path from the root node R to the node with value 38 is R → A → B → C → D.
EXAMPLE #1 – FIND 38

```
3 4 6 9 10 11 12 13 20 22 23 31 35 36 38 41 44
```

R

A

B

C

D

E

F

G

H

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

A

B

C

D

E

F

G

H

I

20

10

35

6

12

23

38

44

3

4

6

9

10

11

12

13

20

22

23

31

35

36

38

41

R

B
EXAMPLE #1 – FIND 38

Diagram of a binary search tree with nodes labeled from 6 to 44, showing the path to find the value 38.
EXAMPLE #1 – FIND 38

A

B

C

D

E

F

G

H

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

```
A
  / 
 10 35
 /   
6 12 23
 /  
3 4 10 12 20 22
 /  
3 4 6 9 10 11 12 13

B
 
C
  / 
38 44
 / 
35 36 38 41
```

W

E

F

G

H

I

3-4-6-9-10-11-12-13-20-22-23-31-35-36-38-41
EXAMPLE #2 – DELETE 38
EXAMPLE #2 – DELETE 38
EXAMPLE #2 – DELETE 38

```
20
  10
     6
     3
   /  \
  4   9
  /    |
10   11
```

```
20
  35
     23
     38
   /   |
  22   44
   /    |
 35   36
```

```
3 4 6 9 10 11 12 13
```

```
E F G H I
```
EXAMPLE #3 – INSERT 45
EXAMPLE #3 – INSERT 45

Diagram of a tree with nodes labeled from 3 to 44, demonstrating the process of inserting the value 45 into the tree.
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

```
    20
   /  
 W   W

  10
 /  
6   12
 /  
3  4  6  9  10  11  12  13
 /  
E  F  G  H  I
```

38/44 (Fall 2021)
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

A

B

C

D

E

F

G

H

I
EXAMPLE #4 – INSERT 25

A

10

B

35

C

23

D

38 44

E

F

G

H

I

6 12

3 4 6 9 10 11 12 13 20 22 23 31 35 36 38 41 44
EXAMPLE #4 – INSERT 25

Diagram showing a binary search tree with nodes labeled from 3 to 44. The tree is structured with nodes containing integers and pointers to other nodes. The diagram illustrates the process of inserting the key 25 into the tree.
EXAMPLE #4 – INSERT 25
EXAMPLE #4 – INSERT 25

Diagram showing a binary search tree with nodes labeled from 3 to 44 and a new node labeled 25 being inserted. The process involves finding the correct position in the tree, where 25 is placed as a new leaf node. The tree structure is shown with nodes labeled A, B, C, D, E, F, G, H, and I, with 25 being added as a new leaf under node D.
EXAMPLE #4 – INSERT 25
EXAMPLE #4 – INSERT 25

A
B
C
D
E
F
G
H
I

20
10
35
6
12
W
W
3
4
6
9
10
11
12
13
20
22
31
38
41
44
Example #4 – Insert 25

We need to split $F$, so we need to hold the latch on its parent node.
We need to split F, so we need to hold the latch on its parent node.
We need to split F, so we need to hold the latch on its parent node.
What was the first step that all the update examples did on the B+Tree?
What was the first step that all the update examples did on the B+Tree?

Delete 38

Insert 45

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.
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

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

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

H will not need to coalesce, so we’re safe!
EXAMPLE #4 – INSERT 25

![Binary Search Tree](image)

- **A**: Root node with value 20
- **B**: Node with value 35
- **C**: Node with value 23
- **D**: Node with value 38
- **E**: Node with value 3
- **F**: Node with value 22
- **G**: Node with value 35
- **H**: Node with value 38
- **I**: Node with value 44
We need to split F, so we have to restart and re-execute like before.
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 we want to move from one leaf node to another leaf node?
LEAF NODE SCAN EXAMPLE #1
LEAF NODE SCAN EXAMPLE #1

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

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

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

**T₁**: Find Keys < 4

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

T₁: Find Keys < 4

Do not release latch on C until thread has latch on B

1 2
B

3 4
C
**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 #2

\[ \text{T}_1: \text{Find Keys} < 4 \]
\[ \text{T}_2: \text{Find Keys} > 1 \]
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

$T_1$: 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

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 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...