09 Index Concurrency Control
Homework #2 is due Sunday Oct 4th

Project #2 is now released:
→ Checkpoint #1: Due Sunday Oct 11th
→ Checkpoint #2: Due Sunday Oct 25th
We assumed that 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?
TODAY'S AGENDA

- Latches Overview
- Hash Table Latching
- B+Tree Latching
- Leaf Node Scans
- Delayed Parent Updates
LOCKS VS. LATCHES

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

Latches
→ Protects 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

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

<table>
<thead>
<tr>
<th>Locks</th>
<th>Latches</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Separate...</strong></td>
<td>User transactions</td>
</tr>
<tr>
<td><strong>Protect...</strong></td>
<td>Database Contents</td>
</tr>
<tr>
<td><strong>During...</strong></td>
<td>Entire Transactions</td>
</tr>
<tr>
<td><strong>Modes...</strong></td>
<td>Shared, Exclusive, Update, Intention</td>
</tr>
<tr>
<td><strong>Deadlock</strong></td>
<td>Detection &amp; Resolution</td>
</tr>
<tr>
<td><strong>...by...</strong></td>
<td>Waits-for, Timeout, Aborts</td>
</tr>
<tr>
<td><strong>Kept in...</strong></td>
<td>Lock Manager</td>
</tr>
<tr>
<td></td>
<td>Threads</td>
</tr>
<tr>
<td></td>
<td>In-Memory Data Structures</td>
</tr>
<tr>
<td></td>
<td>Critical Sections</td>
</tr>
<tr>
<td></td>
<td>Read, Write</td>
</tr>
<tr>
<td></td>
<td>Avoidance</td>
</tr>
<tr>
<td></td>
<td>Coding Discipline</td>
</tr>
<tr>
<td></td>
<td>Protected Data Structure</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 holds the latch 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 Spinlock
Reader-Writer Locks
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;
// Do something special...
m.lock();
// futex
m.unlock();
```
LATCHE 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?
}
```
LATCHE 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:

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

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

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 Locks**

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

---

**Latch**

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

- `write`:
  - `=0`
  - `=0`
Latch Implementations

Choice #3: 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 #3: Reader-Writer Locks
→ Allows for concurrent readers.
→ Must manage read/write queues to avoid starvation.
→ Can be implemented on top of spinlocks.
Choice #3: 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 #3: Reader-Writer Locks**

→ Allows for concurrent readers.
→ Must manage read/write queues to avoid starvation.
→ Can be implemented on top of spinlocks.
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 (i.e., in the header page).
Approach #1: Page Latches
→ Each page has its own reader-write 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.
T₁: Find D

hash(D)
HASH TABLE – PAGE LATCHES

$T_1$: Find D

$\text{hash}(D)$
**HASH TABLE – PAGE LATCHES**

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

$T_2$: Insert E  
$\text{hash}(E)$
HASH TABLE – PAGE LATCHES

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

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

\[ \text{It’s safe to release the latch on Page #1.} \]

T₁: Find D  
\[ \text{hash(D)} \]

T₂: Insert E  
\[ \text{hash(E)} \]

\[ B | \text{val} \]
\[ A | \text{val} \]
\[ C | \text{val} \]
\[ D | \text{val} \]
**HASH TABLE – PAGE LATCHES**

**T₁**: Find D  
\(\text{hash}(D)\)

**T₂**: Insert E  
\(\text{hash}(E)\)

- **Page 0**:  
  - B | val

- **Page 1**:  
  - A | val
  - C | val

- **Page 2**:  
  - D | val
**T₁: Find D**

hash(D)

**T₂: Insert E**

hash(E)

- hash(D)
  - T₁: Find D
  - hash(E)
  - T₂: Insert E

### HASH TABLE – PAGE LATCHES

- B | val
- A | val
- C | val
- D | val

- W
- R

- 0
- 1
- 2
**T₁:** Find D

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

**T₂:** Insert E

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

---

Hash Table - Page Latches

<table>
<thead>
<tr>
<th>Page</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>val</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>val</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>val</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>val</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**HASH TABLE – PAGE LATCHES**

**T₁: Find D**

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

**T₂: Insert E**

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

$T_1$: Find D

$\text{hash}(D)$

$T_2$: Insert E

$\text{hash}(E)$

Diagram:

- Find D
  - $B | \text{val}$
  - $A | \text{val}$
  - $C | \text{val}$
  - $D | \text{val}$

- Insert E
  - $E | \text{val}$
**HASH TABLE – SLOT LATCHES**

T₁: Find D

hash(D)

T₂: Insert E

hash(E)
HA SH TA B LE – SLO T L AT C H ES

T₁: Find D
\(\text{hash}(D)\)

\[\begin{array}{l}
\text{0} \\
B | \text{val} \\
\text{1} \\
A | \text{val} \\
C | \text{val} \\
\text{2} \\
D | \text{val}
\end{array}\]

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

$T_1$: Find D

$\text{hash}(D)$

$T_2$: Insert E

$\text{hash}(E)$

- **Slot 0:**
  - $B | val$

- **Slot 1:**
  - $A | val$
  - $C | val$

- **Slot 2:**
  - $D | val$
HASH TABLE – SLOT LATCHES

T₁: Find D

It’s safe to release the latch on A

hash(D)

T₂: Insert E

hash(E)

R

A | val

B | val

C | val

D | val

W

0

1

2
**T₁: Find D**

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

**T₂: 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

$\text{hash}(E)$
**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)`
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 from 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
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
- **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

T_1: Delete 44
T_2: Find 41

Rebalance!
LATCH CRABBING/COUPLING

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

- Start at node R.
- Compare 38 with 20 (node A), which is greater, so move to the right child (node B).
- Compare 38 with 35 (node B), which is greater, so move to the right child (node D).
- Compare 38 with 44 (node D), which is smaller, so move to the left child (node C).
- Compare 38 with 38 (node C), which matches, so the search is successful.
EXAMPLE #1 – FIND 38

It is now safe to release the latch on A.
EXAMPLE #1 – FIND 38
EXAMPLE #1 – FIND 38
EXAMPLE #1 – FIND 38
EXAMPLE #1 – FIND 38
EXAMPLE #2 – DELETE 38
We may need to coalesce B, so we can’t release the latch on A.
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
EXAMPLE #2 – DELETE 38

```
  20
   / 
 A  10  B
 /  /  /
6 12  35
 /  /  /
3 10 12 23
 /  /  /  /
3 4 6 9 10 11 12 13 20 22 23 31
```

The diagram shows a binary search tree with nodes labeled from 3 to 44. The process of deleting node 38 is illustrated, with node 38 highlighted in red. The deletion involves rebalancing the tree to maintain its properties.
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 #4 – INSERT 25

Diagram of a binary search tree with nodes containing numbers.
EXAMPLE #4 – INSERT 25
EXAMPLE #4 – INSERT 25
EXAMPLE #4 – INSERT 25
EXAMPLE #4 – INSERT 25

A

B

C

D

E

F

G

H

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

- Delete 38
- Insert 45
- Insert 25

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

A

B

C

D

E

F

G

H

I

20

10

35

6

12

23

20

22

35

38

3

4

6

9

10

11

12

13

20

23

31

35

36

38

41

44

44

38

44

38

44

38

44

38

44

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 #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 we want to move from one leaf node to another leaf node?
LEAF NODE SCAN EXAMPLE #1

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

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

\[ T_1: \text{Find Keys < 4} \]

- 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

\( T_i \): Find Keys < 4
LEAF NODE SCAN EXAMPLE #2

$T_1$: Find Keys < 4
$T_2$: Find Keys > 1
LEAF NODE SCAN EXAMPLE #2

T₁: Find Keys < 4
T₂: 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

**T₁**: Find Keys < 4

**T₂**: Find Keys > 1

Only T₁ holds this read latch.

Only T₂ holds this read latch.
LEAF NODE SCAN EXAMPLE #3

T₁: Delete 4
T₂: Find Keys > 1
LEAF NODE SCAN EXAMPLE #3

T₁: Delete 4
T₂: Find Keys > 1
LEAF NODE SCAN EXAMPLE #3

T₁: Delete 4
T₂: 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**

- **$T_1$: 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 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.
Every time a leaf node overflows, we must update at least three nodes.

→ The leaf node being split.
→ The new leaf node being created.
→ The parent node.

**B**\(^\text{link}\)-**T**\(\text{ree Optimization**: When a leaf node overflows, delay updating its parent node.}
EXAMPLE #4 – INSERT 25

$T_1$: Insert 25
EXAMPLE #4 – INSERT 25

$T_1$: Insert 25
EXAMPLE #4 – INSERT 25

$T_1$: Insert 25
EXAMPLE #4 – INSERT 25

$T_1$: Insert 25

Diagram of a B-tree showing the insertion of 25 into a tree with keys 3, 4, 6, 9, 10, 11, 12, 13, 20, 22, 23, 31, 36, 38, 41, 44.
EXAMPLE #4 – INSERT 25

$T_1$: Insert 25

Add the new leaf node as a sibling to $F$, but do not update $C$. 
**EXAMPLE #4 – INSERT 25**

T₁: Insert 25

Add the new leaf node as a sibling to F, but do not update C.
**EXAMPLE #4 – INSERT 25**

$T_1$: Insert 25

Update $C$ the next time that a thread takes a write latch on it.

$\star C$: Add 31
**EXAMPLE #4 – INSERT 25**

- **$T_1$: Insert 25**
- **$T_2$: Find 31**

```plaintext
C: Add 31
```

Diagram:
- Insert 25 at node A
- Find 31 at node C

Nodes:
- 20
- 10
- 6
- 12
- 3
- 4
- 6
- 9
- 10
- 11
- 12
- 13
- 20
- 22
- 23
- 25
- 35
- 36
- 38
- 41
- 44
- 31

Arrows indicate the path taken during insertion and search operations.
EXAMPLE #4 – INSERT 25

$T_1$: Insert 25

$T_2$: Find 31

$\star C$: Add 31
EXAMPLE #4 – INSERT 25

T₁: Insert 25
T₂: Find 31
T₃: Insert 33

C: Add 31
**EXAMPLE #4 – INSERT 25**

- **T₁:** Insert 25
- **T₂:** Find 31
- **T₃:** Insert 33

- **Steps:**
  1. Insert 25
  2. Find 31
  3. Insert 33

- **Note:**
  - **C:** Add 31
EXAMPLE #4 – INSERT 25

\[ T_1: \text{Insert 25} \]
\[ T_2: \text{Find 31} \]
\[ T_3: \text{Insert 33} \]
EXAMPLE #4 – INSERT 25

$T_1$: Insert 25
$T_2$: Find 31
$T_3$: Insert 33

$\star C$: Add 31

Diagram showing the insertion process.
EXAMPLE #4 – INSERT 25

**T₁:** Insert 25

**T₂:** Find 31

**T₃:** Insert 33

- **C:** Add 31
**EXAMPLE #4 – INSERT 25**

- $T_1$: Insert 25
- $T_2$: Find 31
- $T_3$: Insert 33

---

**Graphical Representation**

- **A:** Root node with value 20
- **B:** Node with value 35
- **C:** Starred node with value 31
- **D:** Node with values 38, 44
- **E:** Node with values 35, 36
- **F:** Node with values 38, 41, 44
- **G:** Node with values 31, 33

---

**Operations**

- **T_1:** Insert 25
- **T_2:** Find 31
- **T_3:** Insert 33

---

**Key Notes**

- The diagram illustrates the process of inserting values into a B-tree and finding a specific value, 31.
VERSIONED LATCH COUPLING

Optimistic crabbing scheme where writers are not blocked on readers.

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

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

$T_1$: Find 44
VERSIONED LATCHES: SEARCH 44

$T_1$: Find 44

@A: Read $v^3$
A: Examine Node
VERSIONED LATCHES: SEARCH 44

\( T_1: \text{Find 44} \)

@A
A: Read \( v^3 \)
A: Examine Node

@B
B: Read \( v^5 \)
B: Examine Node

A: Recheck \( v^3 \)

T_1: Find 44
**VERSIONED LATCHES: SEARCH 44**

**$T_1$: Find 44**

- **@A**
  - A: Read $v^3$
  - A: Examine Node

- **@B**
  - B: Read $v^5$
  - A: Recheck $v^3$
  - B: Examine Node

- **@C**
  - C: Read $v^9$
  - B: Recheck $v^5$
  - C: Examine Node
VERSIONED LATCHES: SEARCH 44

T₁: Find 44

A: Read v3
   A: Examine Node

B: Read v5
   B: Examine Node

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

@A

@B

@C

REWIND
**VERSIONED LATCHES: SEARCH 44**

**T₁: Find 44**

@A
- A: Read v3
- A: Examine Node

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

@C
VERSIONED LATCHES: SEARCH 44

T₁: Find 44

@A
A: Read v₃

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

@C
A: Examine Node

D: Read v₄
E: Read v₆
F: Read v₉
VERSIONED LATCHES: SEARCH 44

T₁: Find 44

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

Diagram:

- Node A with value 20
- Node B with value 10
- Node C with value 35
- Node D with value 6
- Node E with value 23
- Node F with value 38

Arrows indicate the path from T₁ to finding the value 44.
VERSIONED LATCHES: SEARCH 44

$T_1$: Find 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**

T₁: Find 44

@A
A: Read v₃
A: Examine Node

B: Read v₅
B: Examine Node

@B
A: Recheck v₃

C: Read v₉

@C
B: Recheck v₅

---

CMU-DB
**VERSIONED LATCHES: SEARCH 44**

**T₁:** Find 44

**T₂:** Insert 31

@A
- A: Read v₃
  - A: Examine Node

@B
- B: Read v₅
  - B: Examine Node

@C
- C: Read v₉
  - C: Recheck v₅

@B
- B: Recheck v₃

@B
- B: Recheck v₅

---

1. Find 44
2. Insert 31
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...
PROJECT #2

You will build a thread-safe B+tree.
→ Page Layout
→ Data Structure
→ STL Iterator
→ Latch Crabbing

We define the API for you. You need to provide the method implementations.

https://15445.courses.cs.cmu.edu/fall2020/project2/
CHECKPOINT #1

Due Date: October 11th @ 11:59pm
Total Project Grade: 40%

Page Layouts
→ How each node will store its key/values in a page.
→ You only need to support unique keys.

Data Structure (Find + Insert)
→ Support point queries (single key).
→ Support inserts with node splitting.
→ Does not need to be thread-safe.
CHECKPOINT #2

Due Date: October 25\textsuperscript{th} @ 11:59pm
Total Project Grade: 60%

Data Structure (Deletion)
→ Support removal of keys with sibling stealing + merging.

Index Iterator
→ Create a STL iterator for range scans.

Concurrent Index
→ Implement latch crabbing/coupling.
DEVELOPMENT HINTS

Follow the textbook semantics and algorithms.

Set the page size to be small (e.g., 512B) when you first start so that you can see more splits/merges.

Make sure that you protect the internal B+Tree root_page_id member.
THINGS TO NOTE

Do **not** change any other files in the system.

Make sure you pull the latest changes from the main BusTub repo.

Post your questions on Piazza or come to TA office hours.
PLAGIARISM WARNING

Your project implementation must be your own work.
→ You may **not** copy source code from other groups or the web.
→ Do **not** publish your implementation on Github.

Plagiarism will **not** be tolerated.
See [CMU's Policy on Academic Integrity](#) for additional information.
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 $V$

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

```c
__sync_bool_compare_and_swap(&M, 20, 30)
```