Index Concurrency Control
ADMINISTRIVIA

Project #1 is due October 2\textsuperscript{nd} @ 11:59pm
→ Special Office Hours: Saturday October 1\textsuperscript{st} @ 3pm-5pm
**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!*
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

<table>
<thead>
<tr>
<th><strong>Locks</strong></th>
<th><strong>Latches</strong></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, 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 IMPLEMENTATIONS

Approach #1: 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: 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 Diagram]

Latch

<table>
<thead>
<tr>
<th></th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td><code>0</code></td>
<td><code>0</code></td>
</tr>
<tr>
<td>Example</td>
<td><code>0</code></td>
<td><code>0</code></td>
</tr>
</tbody>
</table>
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`
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`
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`
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`

Latch

<table>
<thead>
<tr>
<th>read</th>
<th>write</th>
</tr>
</thead>
<tbody>
<tr>
<td>=2</td>
<td>=0</td>
</tr>
<tr>
<td>=1</td>
<td>=1</td>
</tr>
</tbody>
</table>
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.
T₁: Find D

\[\text{hash}(D)\]
**HASH TABLE - PAGE LATCHES**

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

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

Diagram showing a hash table with keys $A$, $B$, $C$, and $D$, and values stored in the corresponding entries.
**T_1:** Find D

hash(D)

**T_2:** Insert E

hash(E)

**HASH TABLE - PAGE LATCHES**
**T₁: Find D**

$hash(D)$

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

**T₂: Insert E**

$hash(E)$
**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 - PAGE LATCHES**

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

**T₁: Find 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(D)**
  - 0
    - B | val
  - 1
    - A | val
    - C | val
  - 2
    - D | val
    - E | val
**HASH TABLE - SLOT LATCHES**

**T₁: Find D**

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

**T₂: 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

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

**T₂:** Insert E

\[ \text{hash}(E) \]
**T₁: Find D**

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

**T₂: Insert E**

\[ \text{hash}(E) \]
**T₁: Find D**

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

*It’s safe to release the latch on A*

**T₂: Insert E**

\[ \text{hash}(E) \]
**T₁: Find D**

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

**T₂: Insert E**

\[\text{hash}(E)\]
**T₁: Find D**

hash(D)

**T₂: Insert E**

hash(E)
**HASH TABLE - SLOT LATCHES**

**$T_1$: Find D**

$\text{hash}(D)$

**$T_2$: Insert E**

$\text{hash}(E)$

- **0**
  - $B | val$

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

- **2**
  - $D | val$
  - $W$
**HASH TABLE - SLOT LATCHES**

<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>
<tr>
<td></td>
<td>E</td>
</tr>
</tbody>
</table>

**T₁: Find D**

`hash(D)`

**T₂: Insert E**

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

\[ \text{T}_1: \text{Find } D \]

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

\[ \text{T}_2: \text{Insert } E \]

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

T₁: Delete 44
T₂: 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
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
It is now safe to release the latch on A.
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
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

Diagram of a binary search tree with nodes containing integers. The tree starts with the root node 20, followed by nodes 10, 6, 3, 4, 6, 9, 10, 11, 12, 13, 20, 22, 23, 31, 35, 36, 38, 41, 44, and 45. The tree is balanced, and the process of inserting 45 into the tree is shown with arrows indicating the path from the root to the new node.
EXAMPLE #3 – INSERT 45

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

A

W

B

C

D

E

F

G

H

I

20

10

6

12

23

20

22

38

44

3

4

6

9

10

11

12

13

23

31

35

36

38

41

44
EXAMPLE #4 – INSERT 25

A

B

20

10

6

3

4

9

10

12

13

22

20

23

31

35

36

38

41

44

D

C

8

W

E

F

G

H

I
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?

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

A

R

20

B

35

C

23

38

D

44

E

3

4

6

9

ger 10

11

12

13

20

22

23

31

35

36

38

41

44

G

H

I
EXAMPLE #2 – DELETE 38
EXAMPLE #2 – DELETE 38

Diagram of a binary search tree with keys 10, 20, 35, 6, 12, 23, 38, 44, 3, 4, 6, 9, 10, 11, 12, 13, 20, 22, 23, 31, 35, 36, 38, 41, 44.
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!
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

T₁: 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

T₁: Find Keys < 4
LEAF NODE SCAN EXAMPLE #2

T₁: Find Keys < 4
T₂: 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 $> 1$
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

\[ T_1: \text{Find Keys} < 4 \]
\[ T_2: \text{Find Keys} > 1 \]

Only \( T_1 \) holds this read latch.

Only \( T_2 \) holds this read latch.
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
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

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

\[ T_1: \text{Delete 4} \]
\[ T_2: \text{Find Keys > 1} \]

- \( T_2 \) 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.
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...