ADMINISTRIVIA

Project #1 is due Fri Sept 27th @ 11:59pm

Homework #2 is due Mon Sept 30th @ 11:59pm

Project #2 will be released Mon Sept 30th
OBSERVATION

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

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

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

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

Latch

<table>
<thead>
<tr>
<th></th>
<th>read</th>
<th>write</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>![Read Symbol] = 0</td>
<td>![Write Symbol] = 0</td>
</tr>
<tr>
<td></td>
<td>![Read Symbol] = 0</td>
<td>![Write Symbol] = 0</td>
</tr>
</tbody>
</table>
**LATCH IMPLEMENTATIONS**

**Approach #3: Reader-Writer Latch**

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

\[
\begin{align*}
\text{read latch} & = 0 \\
\text{write latch} & = 0 \\
\text{Latch} & \quad \text{Lock}
\end{align*}
\]
LATCH IMPLEMENTATIONS

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

Latch

read
\[ \text{read} = 2 \]
\[ \text{write} = 0 \]

write
\[ \text{read} = 0 \]
\[ \text{write} = 0 \]
Approach #3: Reader-Writer Latch
→ Allows for concurrent readers
→ Must manage read/write queues to avoid starvation
→ Can be implemented on top of spinlocks
LATCH IMPLEMENTATIONS

Approach #3: Reader-Writer Latch
→ 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 latch on the entire table (i.e., in the header page).
HASH TABLE LATCHING

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.
**HASH TABLE – PAGE LATCHES**

$T_1$: Find $D$  

$hash(D)$
\textbf{T}_1: \text{Find D} \\
\text{hash}(D)
**HASH TABLE – PAGE LATCHES**

$T_1$: Find D

$hash(D)$

$T_2$: Insert E

$hash(E)$

- $B | val$
- $A | val$
- $C | val$
- $D | val$
**T₁:** Find D  
\[\text{hash}(D)\]

**T₂:** Insert E  
\[\text{hash}(E)\]

**Hash Table – Page Latches**

- **B**: val
- **A**: val
- **C**: val
- **D**: val
HASH TABLE – PAGE LATCHES

$T_1$: Find D
hash(D)

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

$T_2$: Insert E
hash(E)

$B|val$

$A|val$

$C|val$

$D|val$

Find D

Insert E
 HASH TABLE – PAGE LATCHES

$T_1$: Find $D$

$\text{hash}(D)$

$T_2$: Insert $E$

$\text{hash}(E)$
**T₁: Find D**

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

**T₂: Insert E**

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

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

\[ T_2: \text{Insert } E \]
\[ \text{hash}(E) \]
**T₁**: Find D  
\[ \text{hash}(D) \]

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

- \( B \mid \text{val} \)
- \( A \mid \text{val} \)
- \( C \mid \text{val} \)
- \( D \mid \text{val} \)
- \( E \mid \text{val} \)

Hash Table - Page Latches
**HASH TABLE – SLOT LATCHES**

**T₁:** Find D

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

**T₂:** Insert E

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

**$T_1$: Find D**

$hash(D)$

**$T_2$: Insert E**

$hash(E)$

---

Hash Table:

- **Slot 0**: A latch with value $B$ (421, 260)
- **Slot 1**: A latch with value $A$ (421, 153) and a lock symbol, indicating it's locked
- **Slot 2**: A latch with value $D$ (421, 46)

**Key Operations**:

1. **$T_1$: Find D**
   - Map $D$ to Slot 0
2. **$T_2$: Insert E**
   - Map $E$ to Slot 1
\( T_1: \) Find \( D \)

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

\( T_2: \) Insert \( E \)

\( \text{hash}(E) \)
**T₁: Find D**

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

**T₂: Insert E**

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

T₁: Find D

hash(D):

It’s safe to release the latch on A

T₂: Insert E

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

**T₁: Find D**  
\[ \text{hash}(D) \]

**T₂: Insert E**  
\[ \text{hash}(E) \]
**T_1**: Find D  
\[\text{hash}(D)\]

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

**T₁:** Find D

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

**T₂:** Insert E

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

- **T₁:** Find D
  - hash(D)
  - Slot 0
    - B\( | \)val
  - Slot 1
    - A\( | \)val
    - C\( | \)val
  - Slot 2
    - D\( | \)val
    - E\( | \)val

- **T₂:** Insert E
  - hash(E)
  - Slot 2
    - E\( | \)val

**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 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**

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

$T_1$: 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

T_1: Delete 44
T_2: 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!
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
EXAMPLE #1 – FIND 38

It’s safe to release the latch on A.
EXAMPLE #1 – FIND 38

To find the node with the value 38, we start at the root node (A) with the value 20.

- We compare 38 with 20. Since 38 is greater than 20, we move to the right child (B) of the root node.
- At node B, we compare 38 with 35. Since 38 is less than 35, we move to the left child (D) of node B.
- At node D, we compare 38 with 44. Since 38 is less than 44, we move to the left child (G) of node D.
- At node G, we compare 38 with 38. Since they are equal, we stop and the node is found.

The path from the root node (A) to the node with the value 38 is: A → B → D → G.
EXAMPLE #1 – FIND 38
EXAMPLE #1 – FIND 38

A

B

C

D

E

F

G

H

I
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 need to merge with C, so it's safe to release latches on A and B.
EXAMPLE #2 − DELETE 38

We know that D will not need to merge with C, so it’s safe to release latches on A and B.
EXAMPLE #2 – DELETE 38
EXAMPLE #2 – DELETE 38
EXAMPLE #3 – INSERT 45
We know that if D needs to split, B has room so it's safe to release the latch on A.
EXAMPLE #3 – INSERT 45
Example #3 – Insert 45

Node I won’t split, so we can release B+D.
EXAMPLE #3 – INSERT 45

Node I won't split, so we can release B+D.
EXAMPLE #4 – INSERT 25
EXAMPLE #4 – INSERT 25
EXAMPLE #4 – INSERT 25
EXAMPLE #4 – INSERT 25
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.
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?
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.

Can we do better?
BETTER LATCHING ALGORITHM

Assume that the leaf node is safe.

Use read latches and crabbing to reach it, and then verify that it is safe.

If leaf is not safe, then do previous algorithm using write latches.
EXAMPLE #2 – DELETE 38
EXAMPLE #2 – DELETE 38
EXAMPLE #2 – DELETE 38

A

B

C

D

E

F

G

H

I

20

10

6

3

4

6

9

10

11

12

13

20

22

23

31

35

36

38

41

44

35
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.
**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.
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

$T_1$: 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

$T_i$: 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

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

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

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

$T_1$: Delete 4
$T_2$: 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**

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

**Notes**:
- **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

$T_2$ 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.
DELAYED PARENT UPDATES

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\textsuperscript{link}-Tree 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

Add the new leaf node as a sibling to F, but do not update C.
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

T₁: Insert 25

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

C: Add 31
**EXAMPLE #4 – INSERT 25**

**T₁:** Insert 25

**T₂:** Find 31

---

**C:** Add 31
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

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

$\star C$: 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

C: Add 31
EXAMPLE #4 – INSERT 25

**T₁**: Insert 25

**T₂**: Find 31

**T₃**: Insert 33
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...