Index Concurrency Control
Project #1 is due TODAY!

Homework #2 is due Friday Sept 28th @ 11:59pm

Project #2 first checkpoint is due Monday Oct 8th.
OBSERVATION

We assumed that all of 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.
We assumed that all of 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.
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?
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 I see the data that I am supposed to see?

→ **Physical Correctness:** Is the internal representation of the object sound?
TODAY'S AGENDA

Latch Modes
Index Crabbing/Coupling
Leaf Scans
Delayed Parent Updates
LOCKS VS. LATCHES

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

Latches
→ Protects the critical sections of the index’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></th>
<th>Locks</th>
<th>Latches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separate...</td>
<td>User transactions</td>
<td>Threads</td>
</tr>
<tr>
<td>Protect...</td>
<td>Database Contents</td>
<td>In-Memory Data Structures</td>
</tr>
<tr>
<td>During...</td>
<td>Entire Transactions</td>
<td>Critical Sections</td>
</tr>
<tr>
<td>Modes...</td>
<td>Shared, Exclusive, Update, Intention</td>
<td>Read, Write</td>
</tr>
<tr>
<td>Deadlock</td>
<td>Detection &amp; Resolution</td>
<td>Avoidance</td>
</tr>
<tr>
<td>...by...</td>
<td>Waits-for, Timeout, Aborts</td>
<td>Coding Discipline</td>
</tr>
<tr>
<td>Kept in...</td>
<td>Lock Manager</td>
<td>Protected Data Structure</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>Shared, Exclusive,</td>
<td>Avoidance</td>
</tr>
<tr>
<td>Update, Intention</td>
<td>Coding Discipline</td>
</tr>
<tr>
<td>Deadlock Detection &amp;</td>
<td>Protected Data Structure</td>
</tr>
<tr>
<td>Resolution</td>
<td></td>
</tr>
<tr>
<td>...by...</td>
<td></td>
</tr>
<tr>
<td>Waits-for, Timeout,</td>
<td></td>
</tr>
<tr>
<td>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>Lecture 17</th>
<th>Locks</th>
<th>Latches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separate...</td>
<td>User transactions</td>
<td>Threads</td>
</tr>
<tr>
<td>Protect...</td>
<td>Database Contents</td>
<td>In-Memory Data Structures</td>
</tr>
<tr>
<td>During...</td>
<td>Entire Transactions</td>
<td>Critical Sections</td>
</tr>
<tr>
<td>Modes...</td>
<td>Shared, Exclusive, Update, Intention</td>
<td>Read, Write</td>
</tr>
<tr>
<td>Deadlock</td>
<td>Detection &amp; Resolution</td>
<td>Avoidance</td>
</tr>
<tr>
<td>...by...</td>
<td>Waits-for, Timeout, Aborts</td>
<td>Coding Discipline</td>
</tr>
<tr>
<td>Kept in...</td>
<td>Lock Manager</td>
<td>Protected Data Structure</td>
</tr>
</tbody>
</table>

Source: Goetz Graefe
LATCH MODES

Read Mode
→ Multiple threads are allowed to read the same item at the same time.
→ A thread can acquire the read latch if another thread has it in read mode.

Write Mode
→ Only one thread is allowed to access the item.
→ 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>
B+TREE CONCURRENCY CONTROL

We want to allow multiple threads to read and update a B+tree index 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

T₁: Delete 44
B+TREE MULTI-THREADED EXAMPLE

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

\[ T_1: \text{Delete 44} \]
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!
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

**Search:** 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 – SEARCH 38
EXAMPLE #1 – SEARCH 38

It’s safe to release the latch on A.
EXAMPLE #1 – SEARCH 38
EXAMPLE #1 – SEARCH 38
EXAMPLE #1 – SEARCH 38
EXAMPLE #1 – SEARCH 38
EXAMPLE #2 – DELETE 38

W

A

10

20

35

B

C

38

44

D

E

F

G

H

I
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.
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

Diagram showing a binary search tree with keys and the process of deleting the key 38.
EXAMPLE #2 – DELETE 38
EXAMPLE #3 – INSERT 45
EXAMPLE #3 – INSERT 45
EXAMPLE #3 – INSERT 45
I has room so it won't split, so we can release B+D.
EXAMPLE #3 – INSERT 45

I has room so it 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

A

B

C

D

E

F

G

H

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

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

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

What was the first step that all of 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?
Assume that the leaf node is safe. Use read latches and crabbing to reach it, and 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
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

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 of 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

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

$T_1$: Find Keys < 4

Diagram:

- Node A with keys 3
- Node B with keys 1, 2
- Node C with keys 3, 4
- Node R connecting B and C
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

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

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
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

$T_1$: 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

$T_1$: 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. B+tree code must cope with failed latch acquisitions.
Every time a leaf node overflows, we have to update at least three nodes.
→ The leaf node being split.
→ The new leaf node being created.
→ The parent node.

**B**<sub>link</sub>-**Tree Optimization**: When a leaf node overflows, delay updating its parent node.
EXAMPLE #4 – INSERT 25
EXAMPLE #4 – INSERT 25
EXAMPLE #4 – INSERT 25
Add the new leaf node as a sibling to F, but do not update C
EXAMPLE #4 – INSERT 25

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

Update C the next time that a thread takes a write latch on it.
EXAMPLE #4 – INSERT 25

Update C the next time that a thread takes a write latch on it.
Update C the next time that a thread takes a write latch on it.
EXAMPLE #4 – INSERT 25

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

Making a data structure thread-safe seems easy to understand but it is notoriously difficult in practice.

We focused on B+Trees here but the same high-level techniques are applicable to other data structures.
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/fall2018/project2/
CHECKPOINT #1

Due Date: October 8th @ 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 19th @ 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. → See Chapter 15.10

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 file other than the ten that you have to hand it.

We will provide an updated source tarball. You will need to copy over your files from Project #1.

Post your questions on Piazza or come to TA office hours.
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.
We are finally going to discuss how to execute some damn queries…