Lecture #06 – Index Locking & Latching
TODAY’S AGENDA

Index Locks vs. Latches
Latch Implementations
Latch Crabbing
Index Locking
DATABASE INDEX

A data structure that improves the speed of data retrieval operations on a table at the cost of additional writes and storage space.

Indexes are used to quickly locate data without having to search every row in a table every time a table is accessed.
DATA STRUCTURES

Order Preserving Indexes
→ A tree-like structure that maintains keys in some sorted order.
→ Supports all possible predicates with $O(\log n)$ searches.

Hashing Indexes
→ An associative array that maps a hash of the key to a particular record.
→ Only supports equality predicates with $O(1)$ searches.
B-TREE VS. B+TREE

The original **B-tree** from 1972 stored keys + values in all nodes in the tree.
→ More memory efficient since each key only appears once in the tree.

A **B+tree** only stores values in leaf nodes. Inner nodes only guide the search process.
→ Easier to manage concurrent index access when the values are only in the leaf nodes.
OBSERVATION

We already know how to use locks to protect objects in the database.

But we have to treat indexes differently because the physical structure can change as long as the logical contents are consistent.
SIMPLE EXAMPLE

Txn #1: Read ‘22’
SIMPLE EXAMPLE

Txn #1: Read ‘22’
Txn #2: Insert ‘21’
SIMPLE EXAMPLE

Txn #1: Read ‘22’
Txn #2: Insert ‘21’
SIMPLE EXAMPLE

Txn #1: Read ‘22’
Txn #2: Insert ‘21’
SIMPLE EXAMPLE

Txn #1: Read ‘22’
Txn #2: Insert ‘21’
Txn #3: Read ‘22’
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><strong>Locks</strong></th>
<th><strong>Latches</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Separate...</td>
<td>Threads</td>
</tr>
<tr>
<td>Protect...</td>
<td>Database Contents</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>Detection &amp; Resolution</td>
</tr>
<tr>
<td>...by...</td>
<td>Avoidance</td>
</tr>
<tr>
<td>Kept in...</td>
<td>Coding Discipline</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Protected Data Structure</th>
</tr>
</thead>
</table>

*Source: Goetz Graefe*
LOCK-FREE INDEXES

Possibility #1: No Locks
→ Txns don’t acquire locks to access/modify database.
→ Still have to use latches to install updates.

Possibility #2: No Latches
→ Swap pointers using atomic updates to install changes.
→ Still have to use locks to validate txns.
LATCH IMPLEMENTATIONS

Blocking OS Mutex
Test-and-Set Spinlock
Queue-based Spinlock
Reader-Writer Locks

Source: Anastasia Ailamaki
COMPARE-AND-SWAP

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

```
__sync_bool_compare_and_swap(&M, 20, 30)
```
COMPARE-AND-SWAP

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

__sync_bool_compare_and_swap(&M, 20, 30)
Choice #1: Blocking OS Mutex

→ Simple to use
→ Non-scalable (about 25ns per lock/unlock invocation)
→ Example:

```cpp
std::mutex m;

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

m.unlock();
```

`pthread_mutex_t`
LATCH IMPLEMENTATIONS

Choice #2: Test-and-Set Spinlock (TAS)
→ Very efficient (single instruction to lock/unlock)
→ Non-scalable, not cache friendly
→ Example: \texttt{std::atomic<T>}

\begin{verbatim}
std::atomic_flag latch;
:
while (latch.test_and_set(...)) {
    // Yield? Abort? Retry?
}
\end{verbatim}
LATCH IMPLEMENTATIONS

Choice #2: Test-and-Set Spinlock (TAS)
→ Very efficient (single instruction to lock/unlock)
→ Non-scalable, not cache friendly
→ Example: `std::atomic<T>

```cpp
std::atomic_flag latch;

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

`std::atomic<bool>...`
LATCH IMPLEMENTATIONS

Choice #3: Queue-based Spinlock (MCS)
→ More efficient than mutex, better cache locality
→ Non-trivial memory management
→ Example: std::atomic<Latch*>
LATCH IMPLEMENTATIONS

Choice #3: Queue-based Spinlock (MCS)
→ More efficient than mutex, better cache locality
→ Non-trivial memory management
→ Example: `std::atomic<Latch*>`

Mellor-Crummey and Scott

```
Base Latch  CPU1 Latch
next  next

CPU1
```
LATCH IMPLEMENTATIONS

Choice #3: Queue-based Spinlock (MCS)
→ More efficient than mutex, better cache locality
→ Non-trivial memory management
→ Example: `std::atomic<Latch*>`

Mellor-Crummey and Scott
Choice #3: Queue-based Spinlock (MCS)

- More efficient than mutex, better cache locality
- Non-trivial memory management
- Example: `std::atomic<Latch*>`

![Diagram](image-url)
LATCH IMPLEMENTATIONS

Choice #3: Queue-based Spinlock (MCS)
→ More efficient than mutex, better cache locality
→ Non-trivial memory management
→ Example: `std::atomic<Latch*>`

Mellor-Crummey and Scott
Choice #3: Queue-based Spinlock (MCS)
→ More efficient than mutex, better cache locality
→ Non-trivial memory management
→ Example: `std::atomic<Latch*>`
LATCH IMPLEMENTATIONS

Choice #4: Reader-Writer Locks
→ Allows for concurrent readers
→ Have to manage read/write queues to avoid starvation
→ Can be implemented on top of spinlocks
LATCH IMPLEMENTATIONS

Choice #4: Reader-Writer Locks

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

Choice #4: Reader-Writer Locks
→ Allows for concurrent readers
→ Have to manage read/write queues to avoid starvation
→ Can be implemented on top of spinlocks
LATCH IMPLEMENTATIONS

Choice #4: Reader-Writer Locks
→ Allows for concurrent readers
→ Have to manage read/write queues to avoid starvation
→ Can be implemented on top of spinlocks
LATCH IMPLEMENTATIONS

Choice #4: Reader-Writer Locks
→ Allows for concurrent readers
→ Have to manage read/write queues to avoid starvation
→ Can be implemented on top of spinlocks
Choice #4: Reader-Writer Locks

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

Choice #4: Reader-Writer Locks

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

Choice #4: Reader-Writer Locks
→ Allows for concurrent readers
→ Have to manage read/write queues to avoid starvation
→ Can be implemented on top of spinlocks
LATCH IMPLEMENTATIONS

Choice #4: Reader-Writer Locks
→ Allows for concurrent readers
→ Have to manage read/write queues to avoid starvation
→ Can be implemented on top of spinlocks
**LATCH CRABBING**

Acquire and release latches on B+Tree nodes when traversing the data structure.

A thread can release latch on a parent node if its child node considered **safe**.
→ Any node that won’t split or merge when updated.
→ Not full (on insertion)
→ More than half-full (on deletion)
LATCH CRABBING

**Search:** Start at root and go down; repeatedly,
→ Acquire read (R) latch on child
→ Then unlock parent if the child is safe.

**Insert/Delete:** Start at root and go down,
obtaining write (W) latches as needed.
Once child is locked, check if it is safe:
→ If child is safe, release all locks on ancestors.
EXAMPLE #1: SEARCH 23
EXAMPLE #1: SEARCH 23
We can release the latch on A as soon as we acquire the latch for C.
EXAMPLE #1: SEARCH 23

We can release the latch on A as soon as we acquire the latch for C.
We can release the latch on A as soon as we acquire the latch for C.
We can release the latch on A as soon as we acquire the latch for C.
EXAMPLE #2: DELETE 44
EXAMPLE #2: DELETE 44
EXAMPLE #2: DELETE 44

We may need to coalesce C, so we can’t release the latch on A.
EXAMPLE #2: DELETE 44

We may need to coalesce C, so we can’t release the latch on A.

G will not merge with F, so we can release latches on A and C.
EXAMPLE #2: DELETE 44

We may need to coalesce C, so we can’t release the latch on A.

G will not merge with F, so we can release latches on A and C.
EXAMPLE #3: INSERT 40
EXAMPLE #3: INSERT 40
EXAMPLE #3: INSERT 40

C has room if its child has to split, so we can release the latch on A.
EXAMPLE #3: INSERT 40

C has room if its child has to split, so we can release the latch on A.
EXAMPLE #3: INSERT 40

C has room if its child has to split, so we can release the latch on A.

G has to split, so we can’t release the latch on C.
EXAMPLE #3: INSERT 40

C has room if its child has to split, so we can release the latch on A.

G has to split, so we can’t release the latch on C.
OBSERVATION

What was the first step that the DBMS took in the two examples that updated the index?

Delete 44

Insert 40
BETTER LATCH CRABBING

Optimistically assume that the leaf is safe.
→ Take $R$ latches as you traverse the tree to reach it and verify.
→ If leaf is not safe, then do previous algorithm.
EXAMPLE #4: DELETE 44

Diagram showing a B-tree structure with nodes and values.
 EXAMPLE #4: DELETE 44

We assume that C is safe, so we can release the latch on A.
EXAMPLE #4: DELETE 44

We assume that C is safe, so we can release the latch on A.
We assume that C is safe, so we can release the latch on A.

Acquire an exclusive latch on G.
EXAMPLE #4: DELETE 44

We assume that C is safe, so we can release the latch on A.
Acquire an exclusive latch on G.
EXAMPLE #4: DELETE 44

We assume that C is safe, so we can release the latch on A.

Acquire an exclusive latch on G.
OBSERVATION

Crabbing ensures that txns do not corrupt the internal data structure during modifications.

But because txns release latches on each node as soon as they are finished their operations, we cannot guarantee that phantoms do not occur...
PROBLEM SCENARIO #1

A

B

C

D

E

F

G
PROBLEM SCENARIO #1

Txn #1: Check if 25 exists
**PROBLEM SCENARIO #1**

**Txn #1**: Check if 25 exists
PROBLEM SCENARIO #1

Txn #1: Check if 25 exists
**PROBLEM SCENARIO #1**

**Txn #1:** Check if 25 exists
**Problem Scenario #1**

**Txn #1:** Check if 25 exists  
**Txn #2:** Insert 25
**PROBLEM SCENARIO #1**

**Txn #1:** Check if 25 exists  
**Txn #2:** Insert 25
**PROBLEM SCENARIO #1**

**Txn #1:** Check if 25 exists  
**Txn #2:** Insert 25

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>12</th>
<th>23</th>
<th>25</th>
<th>38</th>
<th>44</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td>23</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>38</td>
<td>44</td>
</tr>
</tbody>
</table>

Diagram:

- **A**: 20
- **B**: 10
- **C**: 35
- **D**: 6
- **E**: 12
- **F**: 23, 25
- **G**: 38, 44

Activities:

- **Write (W)**: Lock on E
- **Read (R)**: Check if 25 exists
- **Insert**: Insert 25
PROBLEM SCENARIO #1

Txn #1: Check if 25 exists
Txn #2: Insert 25
Txn #1: Insert 25
PROBLEM SCENARIO #2

Txn #1: Scan [12, 23]
PROBLEM SCENARIO #2

**Txn #1:** Scan [12, 23]
PROBLEM SCENARIO #2

Txn #1: Scan [12, 23]
PROBLEM SCENARIO #2

Txn #1: Scan [12, 23]
Txn #2: Insert 21
PROBLEM SCENARIO #2

**Txn #1:** Scan [12, 23]

**Txn #2:** Insert 21
PROBLEM SCENARIO #2

Txn #1: Scan [12, 23]
Txn #2: Insert 21
PROBLEM SCENARIO #2

Txn #1: Scan [12, 23]
Txn #2: Insert 21
PROBLEM SCENARIO #2

**Txn #1:** Scan [12, 23]

**Txn #2:** Insert 21

**Txn #1:** Scan [12, 23]
**PROBLEM SCENARIO #2**

**Txn #1:** Scan [12, 23]  
**Txn #2:** Insert 21  
**Txn #1:** Scan [12, 23]
INDEX LOCKS

Need a way to protect the index's logical contents from other txns to avoid phantoms.

Difference with index latches:
→ Locks are held for the entire duration of a txn.
→ Only acquired at the leaf nodes.
→ Not physically stored in index data structure.
INDEX LOCKS

Lock Table

txn1: X
txn2: S
txn3: S
• • •

txn3: S
txn2: S
txn4: S
• • •

txn4: IX
txn6: X
txn5: S
• • •
INDEX LOCKS

Lock Table

- **txn1**: X
- **txn2**: S
- **txn3**: S
- **txn4**: S
- **txn5**: S
- **txn6**: X
INDEX LOCKING SCHEMES

Predicate Locks
Key-Value Locks
Gap Locks
Key-Range Locks
Hierarchical Locking
PREDICATE LOCKS

Proposed locking scheme from System R.
→ Shared lock on the predicate in a WHERE clause of a SELECT query.
→ Exclusive lock on the predicate in a WHERE clause of any UPDATE, INSERT, or DELETE query.

Never implemented in any system.
SELECT SUM(balance) FROM account WHERE name = 'Biggie'

INSERT INTO account (name, balance) VALUES ('Biggie', 100);
SELECT SUM(balance) FROM account WHERE name = 'Biggie'

INSERT INTO account (name, balance) VALUES ('Biggie', 100);
KEY-VALUE LOCKS

Locks that cover a single key value.
Need “virtual keys” for non-existent values.

B+Tree Leaf Node

10 12 14 16
KEY-VALUE LOCKS

Locks that cover a single key value. Need “virtual keys” for non-existent values.
GAP LOCKS

Each txn acquires a key-value lock on the single key that it wants to access. Then get a gap lock on the next key gap.

B+Tree Leaf Node

10  12  14  16
GAP LOCKS

Each txn acquires a key-value lock on the single key that it wants to access. Then get a gap lock on the next key gap.

B+Tree Leaf Node

| 10 | {Gap} | 12 | {Gap} | 14 | {Gap} | 16 |
GAP LOCKS

Each txn acquires a key-value lock on the single key that it wants to access. Then get a gap lock on the next key gap.
KEY-RANGE LOCKS

Atxn takes locks on ranges in the key space.
→ Each range is from one key that appears in the relation, to the next that appears.
→ Define lock modes so conflict table will capture commutativity of the operations available.
KEY-RANGE LOCKS

Locks that cover a key value and the gap to the next key value in a single index.
→ Need “virtual keys” for artificial values (infinity)

B+Tree Leaf Node

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Gap</td>
<td>12</td>
<td>Gap</td>
</tr>
</tbody>
</table>
KEY-RANGE LOCKS

Locks that cover a key value and the gap to the next key value in a single index.
→ Need “virtual keys” for artificial values (infinity)

B+Tree Leaf Node

10 {Gap} 12 {Gap} 14 {Gap} 16

Next Key [14, 16]
KEY-RANGE LOCKS

Locks that cover a key value and the gap to the next key value in a single index.
→ Need “virtual keys” for artificial values (infinity)

B+Tree Leaf Node

10  {Gap}  12  {Gap}  14  {Gap}  16

Prior Key (12, 14)
HIERARCHICAL LOCKING

Allow for a txn to hold wider key-range locks with different locking modes.
→ Reduces the number of visits to lock manager.

B+Tree Leaf Node

10 {Gap} 12 {Gap} 14 {Gap} 16
HIERARCHICAL LOCKING

Allow for a txn to hold wider key-range locks with different locking modes.
→ Reduces the number of visits to lock manager.
HIERARCHICAL LOCKING

Allow for a txn to hold wider key-range locks with different locking modes.
→ Reduces the number of visits to lock manager.
HIERARCHICAL LOCKING

Allow for a txn to hold wider key-range locks with different locking modes.
→ Reduces the number of visits to lock manager.
Hierarchical locking essentially provides predicate locking without complications.
→ Index locking occurs only in the leaf nodes.
→ Latching is to ensure consistent data structure.

Peloton currently does not support serializable isolation with range scans.
NEXT CLASS

Index Key Representation
Memory Allocation & Garbage Collection
T-Trees (1980s / TimesTen)
Concurrent Skip Lists (MemSQL)