

Lecture #10: Index Concurrency Control

15-445/645 Database Systems (Spring 2026)

<https://15445.courses.cs.cmu.edu/spring2026/>

Carnegie Mellon University

Andy Pavlo

1 Index Concurrency Control

So far, we assumed that the data structures we have discussed are single-threaded. However, most DBMSs need to allow multiple threads to safely access data structures to take advantage of additional CPU cores and hide disk I/O stalls.

There are some systems that use a single-threaded model as some arguments say that it would be faster without latching (e.g. Redis). Additionally, an easy way to convert a single-threaded data structure to a multi-threaded one is to use a single read-write lock to guard the entire structure, but this is not an efficient method.

A *concurrency control* protocol is the method that the DBMS uses to ensure “correct” results for concurrent operations on a shared object from multiple workers. We use the term “workers” here to more generically describe multiple concurrent accessors: some DBMSs use processes (e.g. Postgres) while others use threads, etc.

A protocol’s correctness criteria can vary:

- **Logical Correctness:** This means that the worker is able to read values that it expects to read, e.g. a thread should read back the value it had written previously.
- **Physical Correctness:** This means that the internal representation of the object is sound, e.g. there are no pointers in the data structure that will cause a worker to read invalid memory locations.

For the purposes of this lecture, we only care about enforcing physical correctness. We will revisit logical correctness in later lectures.

2 Locks vs. Latches

There is an important distinction between locks and latches when discussing how the DBMS protects its internal elements.

Locks

A lock is a higher-level, logical primitive that protects the contents of a database (e.g., tuples, tables, databases) from other transactions. Transactions will typically hold a lock for its entire duration. Database systems can expose to the user the locks that are being held as queries are run. There should be some higher-level mechanism to detect deadlocks and rollback changes.

Latches

Latches are the low-level protection primitives used for critical sections the DBMS’s internal data structures (e.g., data structure, regions of memory) from other workers. Latches are held for a short period for a simple operation in a database system (i.e., page latch). Unlike with locks, it is the worker’s responsibility to avoid deadlocks / roll back changes (rather than another mechanism within the system).

There are two modes for latches:

- **READ:** Multiple workers are allowed to read the same item at the same time. A worker can acquire the latch in read mode even if another thread has already acquired it in read mode.
- **WRITE:** Only one worker is allowed to access the item. A worker cannot acquire a write latch if another thread holds the latch in any mode. A worker holding a write latch also prevents other worker from acquiring a read latch.

3 Latch Implementations

There are some key goals we want to achieve when implementing latches:

- Small memory footprint (ideally measured in bytes)
- Fast execution path when there is no contention
- Decentralised management of latches
- Avoid expensive system calls

The underlying primitive that used to implement a latch is through atomic instructions that modern CPUs provide. With this, a thread can check the contents of a memory location to see whether it has a certain value.

- **Atomic Instruction Example: compare-and-swap (CAS)**
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

There are several approaches to implementing a latch in a DBMS. Each approach has different trade-offs in terms of engineering complexity and runtime performance. These test-and-set steps are performed atomically (i.e., no other thread can update the value in between the test and set steps).

Test-and-Set Spin Latch (TAS)

Spin latches are a more efficient alternative to an OS mutex as it is controlled by the DBMSs. A spin latch is essentially a location in memory that threads try to update (e.g., setting a boolean value to true). A thread performs CAS to attempt to update the memory location. The DBMS can control what happens if it fails to get the latch. It can choose to try again (for example, using a while loop) or allow the OS to deschedule it. Thus, this method gives the DBMS more control than the OS mutex, where failing to acquire a latch gives control to the OS.

- **Example:** `std::atomic<T>`
- **Advantages:** Latch/unlatch operations are efficient (single instruction to lock/unlock on x86).
- **Disadvantages:** Not scalable nor cache-friendly because with multiple threads, the CAS instructions will be executed multiple times in different threads. These wasted instructions will pile up in high contention environments; the threads look busy to the OS even though they are not doing useful work. Furthermore, in a non-uniform memory architecture polling the memory location of the latch may be especially expensive if it is located in the local cache/memory of another CPU.

Blocking OS Mutex

One possible implementation of latches is the OS built-in mutex infrastructure. Linux provides the `futex` (fast user-space mutex), which is comprised of (1) a spin latch in user-space and (2) an OS-level mutex. If the DBMS can acquire the user-space latch, then the latch is set. It appears as a single latch to the DBMS even though it contains two internal latches. If the DBMS fails to acquire the user-space latch, then it goes

down into the kernel and tries to acquire a more expensive mutex. If the DBMS fails to acquire this second mutex, the OS de-schedules the thread and notifies the thread when the mutex becomes available.

OS mutex is generally a bad idea inside of DBMSs as it is managed by OS and has large overhead.

- **Example:** `std::mutex`
- **Advantages:** Simple to use and requires no additional coding in DBMS.
- **Disadvantages:** Expensive and non-scalable (about 25 ns per lock/unlock invocation) because of OS scheduling. OS has to do its own bookkeeping with its own latching to manage which threads should be asleep / which should be notified.

Reader-Writer Latches

Mutexes and Spin Latches do not differentiate between reads/writes (i.e., they do not support different modes). The DBMS needs a way to allow for concurrent reads, so if the application has heavy reads it will have better performance because readers can share resources instead of waiting.

A Reader-Writer Latch allows a latch to be held in either read or write mode. It keeps track of how many threads hold the latch and are waiting to acquire the latch in each mode. Reader-writer latches use one of the previous two latch implementations as primitives and have additional logic to handle reader-writer queues, which are queues requests for the latch in each mode. Different DBMSs can have different policies for how it handles the queues.

One thing to notice is that different reader-writer lock implementations have different waiting policies. There are reader-preferred, writer-preferred, and fair reader-writer locks. The behavior differs in different operating systems and pthread implementations.

- **Example:** `std::shared_mutex`
- **Advantages:** Allows for concurrent readers.
- **Disadvantages:** The DBMS has to manage read/write queues to avoid starvation. Larger storage overhead than spin Latches due to additional meta-data.

4 Hash Table Latching

It is easy to support concurrent access in a static hash table due to the limited ways threads access the data structure. For example, all threads move in the same direction when moving from slot to the next (i.e., top-down). Threads also only access a single page/slot at a time. Thus, deadlocks are not possible in this situation because no two threads could be competing for latches held by the other. When we need to resize the table, we can just take a global latch on the entire table to perform the operation.

Latching in a dynamic hashing scheme (e.g., extendible) is a more complicated scheme because there is more shared state to update, but the general approach is the same.

There are two approaches to support latching in a hash table that differ on the granularity of the latches:

- **Page Latches:** Each page/block has its own Reader-Writer latch that protects its entire contents. Threads acquire either a read or write latch before they access a page. This decreases parallelism because potentially only one thread can access a page at a time, but accessing multiple slots in a page will be fast for a single thread because it only has to acquire a single latch.
- **Slot Latches:** Each slot has its own latch. This increases parallelism because two threads can access different slots on the same page. But it increases the storage and computational overhead of accessing the table because threads have to acquire a latch for every slot they access, and each slot has to store data for the latches. The DBMS can use a single mode latch (i.e., Spin Latch) to reduce meta-data and

computational overhead at the cost of some parallelism.

It is also possible to create a latch-free linear probing hash table directly using compare-and-swap (CAS) instructions. Insertion at a slot can be achieved by attempting to compare-and-swap a special “null” value with the tuple we wish to insert. If this fails, we can probe the next slot, continuing until it succeeds.

5 B+Tree Latching

The challenge of B+Tree latching is preventing the two following 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.

Latch crabbing/coupling is a protocol to allow multiple threads to access/modify B+Tree at the same time. The basic idea is as follows:

1. Get latch for the parent.
2. Get latch for the child.
3. Release latch for the parent if the child is deemed “safe”. A “safe” node is one that will not split, merge, or redistribute when updated. In other words, a node is “safe” if
 - **for insertion:** it is not full.
 - **for deletion:** it is more than half full.

Note that read operations do not need to worry about the “safe” condition (as read operations will not change the size of any node in the tree).

Basic Latch Crabbing Protocol:

- **Find:** Start at the root and go down, repeatedly acquire latch on the child and then unlatch parent.
- **Insert/Delete:** Start at the root and go down, obtaining X latches as needed. Once the child is latched, check if it is safe. If the child is safe, release latches on all its ancestors.

The order in which latches are released is not important from a correctness perspective. However, from a performance point of view, it is better to release the latches that are higher up in the tree since they block access to a larger portion of leaf nodes.

Optimistic Latch Crabbing Protocol: The problem with the basic latch crabbing algorithm is that transactions **always** acquire an exclusive latch on the root for every insert/delete operation. This limits parallelism. Instead, one can assume that having to resize (i.e., split/merge nodes) is rare, and thus transactions can acquire shared latches down to the leaf nodes. Each transaction will assume that the path to the target leaf node is safe, and use READ latches and crabbing to reach it and verify. If the leaf node is not safe, then we abort and do the previous algorithm where we acquire WRITE latches.

- **Find:** Same algorithm as before.
- **Insert/Delete:** Set READ latches as if for find, go to leaf, and set WRITE latch on leaf. If the leaf is not safe, release all previous latches, and restart the transaction using the original Insert/Delete protocol.

Some latch implementations allow for upgrading a read latch to a write latch while holding the latch. In this way it may be possible to not restart the entire traversal of the B+tree when a leaf is not safe, but it is less straightforward and possibly requires already holding some ancestor read latches.

Leaf Node Scans

The threads in these protocols acquire latches in a “top-down” manner. This means that 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. Given this, there can never be deadlocks.

However, leaf node scans are susceptible to deadlocks because now we have threads trying to acquire exclusive locks in two different directions at the same time (e.g., thread 1 tries to delete, thread 2 does a leaf node scan). Index latches do not support deadlock detection or avoidance.

Thus, the only way programmers can deal with this problem is through coding discipline. The leaf node sibling latch acquisition protocol must support a “no-wait” mode. That is, the B+tree code must cope with failed latch acquisitions. Since latches are intended to be held (relatively) briefly, if a thread tries to acquire a latch on a leaf node but that latch is unavailable, then it should abort its operation (releasing any latches that it holds) quickly and restart the operation. Some trade-off could be made here is to wait a little more before restarting itself, the wait time can depend on how much work has been done so far for the thread.

In the case the index is a high-contention data structure where a leaf scan thread could possibly fail consistently, should there be a mechanism to solve this, a higher level system thread scheduler detecting this and trying to schedule out other threads to get the leaf scan thread done. This mechanism does not necessarily need to be incorporated in a data structure’s implementation but should rely on a higher level part of the system.