Lecture #09: Index Concurrency Control

15-445/645 Database Systems (Fall 2019)
`https://15445.courses.cs.cmu.edu/fall2019/`
Carnegie Mellon University
Prof. Andy Pavlo

1 Index 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? This means that the thread is able to read values that it should be allowed to read.
- **Physical Correctness:** Is the internal representation of the object sound? This means that there are not pointers in our data structure that will cause a thread to read invalid memory locations.

The logical contents of the index is the only thing we care about in this lecture. They are not quite like other database elements so we can treat them differently.

2 Locks vs. Latches

Locks:

- Protects the indexs logical contents from other transactions.
- Held for (mostly) the entire duration of the transaction.
- The DBMS needs to be able to rollback changes.

Latches:

- Protects the critical sections of the indexs internal data structure from other threads.
- Held for operation duration.
- The DBMS does not need to be able to rollback changes.
- Two Modes:
  - **`READ`:** 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`:** 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.

3 Latch Implementations

The underlying primitive that we can use to implement a latch is through an atomic *compare-and-swap* (CAS) instruction that modern CPUs provide. With this, a thread can check the contents of a memory location to see whether it has a certain value. If it does, then the CPU will swap the old value with a new one. Otherwise the memory location remains unmodified.

There are several approaches to implementing a latch in a DBMS. Each approach have 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 after one thread checks it but before it updates it).

Blocking OS Mutex

Use the OS built-in mutex infrastructure as a latch. The futex (fast user-space mutex) is comprised of (1) a spin latch in user-space and (2) a 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, then the thread notifies the OS that it is blocked on the lock and then it is descheduled.

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.

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. If it cannot, then it spins in a while loop forever trying to update it.

- **Example:** `std::atomic<T>`
- **Advantages:** Latch/unlatch operations are efficient (single instruction to lock/unlock).
- **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. This leads to cache coherence problems because threads are polling cache lines on other CPUs.

Reader-Writer Latches

Mutexes and Spin Latches do not differentiate between reads / writes (i.e., they do not support different modes). We need 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.

- **Example:** This is implemented on top of Spin Latches.
- **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. To resize the table, take a global latch on the entire table (i.e., in the header page).

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

In general, there are two approaches to support latching in a hash table:

- **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. 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 because a thread 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 in 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. The DBMS can use a single mode latch (i.e., Spin Latch) to reduce meta-data and computational overhead.

5 B+Tree Latching

Lock crabbing / coupling is a protocol to allow multiple threads to access/modify B+Tree at the same time:

1. Get latch for parent.
2. Get latch for child.
3. Release latch for parent if it is deemed safe. A **safe node** is one that will not split or merge when updated (not full on insertion or more than half full on deletion).

Basic Latch Crabbing Protocol:

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

Improved Lock 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, we 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 any node in the path is not safe, then do previous algorithm (i.e., acquire `WRITE` latches).

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

6 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.

Leaf node scans are susceptible to deadlocks because now we have threads trying to acquire locks in two different directions at the same time (i.e., left-to-right and right-to-left). Index latches do not support deadlock detection or avoidance.

Thus, 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. That is, B+tree code must cope with failed latch acquisitions. This means that if a thread tries to acquire a latch on a leaf node but that latch is unavailable, then it will immediately abort its operation (releasing any latches that it holds) and then restart the operation.