Lecture #7: Index Locking and Latching

15-721 Advanced Database Systems (Spring 2018)
`http://15721.courses.cs.cmu.edu/spring2018/`
Carnegie Mellon University
Prof. Andy Pavlo

Database Indexes

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. Indexes require different locking because the physical structure can change as long as the logical contents are consistent.

Order Preserving Indexes

- A tree 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 support equality predicates with O(1) searches.

B-tree vs B+Tree

- The original **B-Tree** from 1972 stored values in all nodes in the tree [2].
- B-Tree was more memory efficient since each key only appears once in the tree.
- **B+Tree** only stores values in leaf nodes, and inner nodes only guide the search process.
- In practice, people use B+Trees over B-Trees because its easier to manage concurrent index access because values are only in the leaf nodes.

Lock-Free Indexes Approaches

When somebody says that they have a "lock-free" index, it can mean one of two things [3].

No Locks

- Transactions do not acquire locks to access/modify database.
- Still have to use latches to install updates.

No Latches

- Swap pointers using atomic updates to install changes.
- Still have to use locks to validate transactions.

Latch Implementations

Compare and Swap: Atomic instructuion that compares contents of a memory location M to a given value V.

Blocking OS Mutex

- Simple to use.
- Non-scalable (about 25 ns per lock/unlock invocation).
- Example: `std::mutex`

Test-and-Set Spin Lock (TAS)

- Very efficient (single instruction to lock/unlock).
- Non-scalable, not cache friendly.
- Example: `std::atomic<T>`

Queue-Based Spin Lock (MCS)

- Also known as Mellor-Crummey and Scott (MCS) locks.
- More efficient than mutex, better cache locality. But has non-trivial memory management.
- Example: std::atomic<Latch*>

Reader-Writer Locks

- Allows for concurrent readers.
- Have to manage read/write queues to avoid starvation.
- Can be implemented ontop of spin locks.

Index Latching

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 tis child node is considered **safe**:
  - A node is safe it wont split or merge when updated.
  - Not full (on insertion).
  - More than half-full (on deletes).
- **Search:** Start at root and go down, repeatedly acquiring read (R) latch on child, and next unlocking parent.
- **Insert/Delete:** Start at root and go down, acquiring write (W) latches if needed. Once child is locked, if it is safe, release all locks on ancestors

Better Latch Crabbing

The problem with the previous latch crabbing approach is that it requires each thread to lock the root as the first step each time. This is a major bottleneck.

A better approach is to optimistically assume that the leaf is safe [1].

- Take R latches as you traverse the tree to reach leaf and verify.
- If leaf is not safe, then fallback to previous algorithm.

Index Locking Schemes

Crabbing does not protect from phantoms because we are releasing locks as soon as insert/delete operation ends. There needs to be a way to protect the index's logical contents from other transactions to avoid phantoms.

Difference with index latches:

- Locks are held for the entire duration of a transaction.
- Only acquired at the leaf nodes.
- Not physically stored in index data structure.

Predicate Locks

Proposed locking scheme from **IBM System R** [4]. But not used in practice and never implemented in any system.

- 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**, and **DELETE**.
- Precision locks are a simplification of predicate locks.
- Can determine if there will be a conflict by looking at the query without having to run it.

Key-Value Locks

- Locks that cover a single key value.
- Need "virtual keys" for non-existent values.
- Cannot store lock in index.

Gap Locks

- Each transaction acquires a key-value lock on the single key that it wants to access, then get a gap lock on the next key gap.
- The DBMS cannot store lock in an index node because the physical location of a node can change. This means that the DBMS has to scan all of the index nodes to find the locks that a transaction holds in order to release them.

Key-Range Locks

- A transaction 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 of the operations available.

Hierarchical Locking

- Allow for a transaction to hold wider key-range locks with different locking modes.
- Reduces the number of visits to lock manager.
- Allows for nesting of compatible locks (e.g., an X lock inside an IX key-range lock).

References

[1] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Act Informatica*, 9(1):1–21, March 1977. URL `https://dl.acm.org/citation.cfm?id=2699553`.

[2] D. Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979. doi: 10.1145/356770.356776. URL `http://doi.acm.org/10.1145/356770.356776`.

[3] G. Graefe. A survey of b-tree locking techniques. *ACM Trans. Database Syst.*, 35(3):16:1–16:26, July 2010. doi: 10.1145/1806907.1806908. URL `http://doi.acm.org/10.1145/1806907.1806908`.

[4] E. K.P., J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, Nov 1976. URL `https://dl.acm.org/citation.cfm?id=360369`.