Lecture #17: Two-Phase Locking

15-445/645 Database Systems (Fall 2018)
https://15445.courses.cs.cmu.edu/fall2018/
Carnegie Mellon University
Prof. Andy Pavlo

1 Transaction Locks

The DBMS contains a centralized lock manager that decides decisions whether a transaction can have a lock or not. It has a global view of whats going on inside the system.

- **Shared Lock (`S-LOCK`):** A lock that allows multiple transactions to read the same object at the same time. If one transaction holds a shared lock, then another transaction can acquire that same shared lock.
- **Exclusive Lock (`X-LOCK`):** Allows a transaction to modify an object. This lock is not compatible for any other lock. Only one transaction can hold an exclusive lock at a time.

Executing with locks:

1. Transactions request locks (or upgrades) from the lock manager.
2. The lock manager grants or blocks requests based on what locks are currently held by other transactions.
3. Transactions release locks when they no longer need them.
4. The lock manager updates its internal lock-table and then gives locks to waiting transactions.

2 Two-Phase Locking

Two-Phase locking (2PL) is a pessimistic concurrency control protocol that determines whether a transaction is allowed to access an object in the database on the fly. The protocol does not need to know all of the queries that a transaction will execute ahead of time.

Phase #1: Growing

- Each transaction requests the locks that it needs from the DBMS's lock manager.
- The lock manager grants/denies lock requests.

Phase #2: Shrinking

- The transaction enters this phase immediately after it releases its first lock.
- The transaction is allowed to only release locks that it previously acquired. It cannot acquire new locks in this phase.

On its own, 2PL is sufficient to guarantee conflict serializability. It generates schedules whose precedence graph is acyclic. But it is susceptible to *cascading aborts*, which is when a transaction aborts and now another transaction must be rolled back, which results in wasted work.

There are also potential schedules that are serializable but would not be allowed by 2PL (locking can limit concurrency).

3 Strict Two-Phase Locking

The transaction only releases locks when it finishes. There really isn't a shrinking phase like in regular 2PL. A schedule is *strict* if a value written by a transaction is not read or overwritten by other transactions until that transaction finishes.

The advantage of this approach is that the DBMS does not incur cascading aborts. The DBMS can also reverse the changes of an aborted transaction by just restoring original values of modified tuples.

4 2PL Deadlock Handling

A *deadlock* is a cycle of transactions waiting for locks to be released by each other. There are two approaches to handling deadlocks in 2PL: detection and prevention.

Approach #1: Deadlock Detection

The DBMS creates a *waits-for* graph: Nodes are transactions, and edge from T_i to T_j if transaction T_i is waiting for transaction T_j to release a lock. The system will periodically check for cycles in waits-for graph and then make a decision on how to break it.

- When the DBMS detects a deadlock, it will select a "victim" transaction to rollback to break the cycle.
- The victim transaction will either restart or abort depending on how the application invoked it
- There are multiple transaction properties to consider when selecting a victim. There is no one choice that is better than others. 2PL DBMSs all do different things:
  1. By age (newest or oldest timestamp).
  2. By progress (least/most queries executed).
  3. By the # of items already locked.
  4. By the # of transactions that we have to rollback with it.
  5. # of times a transaction has been restarted in the past
- **Rollback Length:** After selecting a victim transaction to abort, the DBMS can also decide on how far to rollback the transaction's changes. Can be either the entire transaction or just enough queries to break the deadlock.

Approach #2: Deadlock Prevention

When a transaction tries to acquire a lock, if that lock is currently held by another transaction, then perform some action to prevent a deadlock. Assign priorities based on timestamps (e.g., older means higher priority). These schemes guarantee no deadlocks because only one type of direction is allowed when waiting for a lock. When a transaction restarts, its (new) priority is its old timestamp.

- **Wait-Die ("Old waits for Young"):** If T_1 has higher priority, T_1 waits for T_2. Otherwise T_1 aborts
- **Wound-Wait ("Young waits for Old"):** If T_1 has higher priority, T_2 aborts. Otherwise T_1 waits.

5 Lock Granularities

If a transaction wants to update a billion tuples, it has to ask the DBMS's lock manager for a billion locks. This will be slow because we have to take latches in the lock manager's internal lock table data structure. Instead, we want to use a lock hierarchy that allow a transaction to take more coarse-grained locks in the system. Acquiring a lock for something in this hierarchy implicitly acquires a lock for all its children.

Intention locks allow a higher level node to be locked in shared or exclusive mode without having to check all descendant nodes. If a node is in an intention mode, then explicit locking is being done at a lower level

in the tree.

- **Intention-Shared (`IS`):** Indicates explicit locking at a lower level with shared locks.
- **Intention-Exclusive (`IX`):** Indicates explicit locking at a lower level with exclusive or shared locks.
- **Shared+Intention-Exclusive (`SIX`):** The subtree rooted at that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.

6 Conclusion

The application does not typically set locks manually. But sometimes it can provide the DBMS with hints to help it improve concurrency:
`SELECT...FOR UPDATE`: Perform a select and then sets an exclusive lock on fetched tuples

2PL is used in almost all DBMS. It automatically provides correct interleavings of transaction operations. But it must handle deadlocks.