Concurrent Control Techniques

Chapter 22

Review: ACID Properties of Transactions

- Transaction execution must maintain the correctness of the database model
- Therefore additional requirements are placed on the execution of transactions beyond those placed on ordinary programs
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability

Serializable Schedules

- The concurrent schedule *S:* $r_1(x)\ w_2(z)\ w_1(y)$ is equivalent to the serial schedules of *T1* and *T2* in either order:
  - *T1, T2:* $r_1(x)\ w_1(y)\ w_2(z)$ and
  - *T2, T1:* $w_2(z)\ r_1(x)\ w_1(y)$

since operations of distinct transactions on different data items commute. Hence, *S* is a serializable schedule

Classification of CCTs

- 22.1 2PL
  - lock types, basic/2PL/strict
  - deadlocks
- 22.2 Timestamp Ordering
  - timestamps, ordering algo
- 22.3 Multiversion CCT
  - w time-stamps, 2-phase
- 22.4 Validation (Optimistic) based CCT

What is concurrent control?

- Def.: Protocols, Algorithms, Techniques to avoid problems
  - resulting from concurrent operations on shared data
  - Problems of lost update, incorrect summary …
- A Well-known problem in O.S., Client-Server, Shared Resource
  - Ex. Concurrent processes/threads in Operating Systems
    - Mutual Exclusion Protocols: semaphores, critical sections

Common Concurrency Control Techniques (CCTs)

- Goal:
  - Quickly ensure serializability
  - Fast filter to avoid violation of serializability
- may rule many possible serializable schedule
- Prescribes a Protocol to be observed by each transaction
- < current state, new op. request > → grant/delay op, abort T

Simple CCTs

- 2 phase locking (2PL) protocol for transactions
  - Schedule: order Ti by the order of lock acquisition
- TSO: item-based violation test at read/write
  - Schedule: order Ti by their time-stamps
- *Other schemes*
  - Optimistic, 3 phases (read, validate, write)
    - Validate phase = conflict? w/committed/validated Tj
  - Multiversion - uses version semantics

Lock Based CCT : Basic Definitions

- *LOCK[X] = a variable describing status of data item X*
  - wrt possible applicable operations without conflict.
- *Lock(X, op)= permission to Ti to perform op on X*
  - NOTE Convention: LOCK(X) = variable, Lock(X) = permission
- *Operations on LOCK(X) requested by Ti to Lock Manager*
  - lock_item(X), lock_item(X, read), unlock(X), ...
  - Protocol: Ti needs Lock(X, op) to perform op(X)
  - Lock Manager: (operation on LOCK(X)) → { grant, postpone, deny }
  - See Fig. 22.1, 22.2 (pp 779-781) for algorithms

Types of Locks

- binary locks:
  - Binary values (states): unlock= 0, lock=1
  - ops.: lock_item(X), unlock_item(X))
  - See protocol rules on pp 779
- Read/Write (Shared/Exclusive) locks:
  - 3 Values:
    - unlocked, read_locked (shared_locked), write_locked(exclusive_locked).
  - 3 ops:
    - read_lock(X) is shared: Multiple read locks can be allowed on X.
    - write_lock(X) exclusive, no other concurrent lock on X
    - unlock(X)
  - See pp779-782 the list of locking rules
- Conversion of Locks
  - Relax rule 4 and 5
  - Can convert the lock from one lock state to another
    - Upgrade lock: read_lock(X) → write_lock(X)
    - Downgrade: write_lock(X) → read_lock(X)

Implementing Serializability: Two-Phase Locking

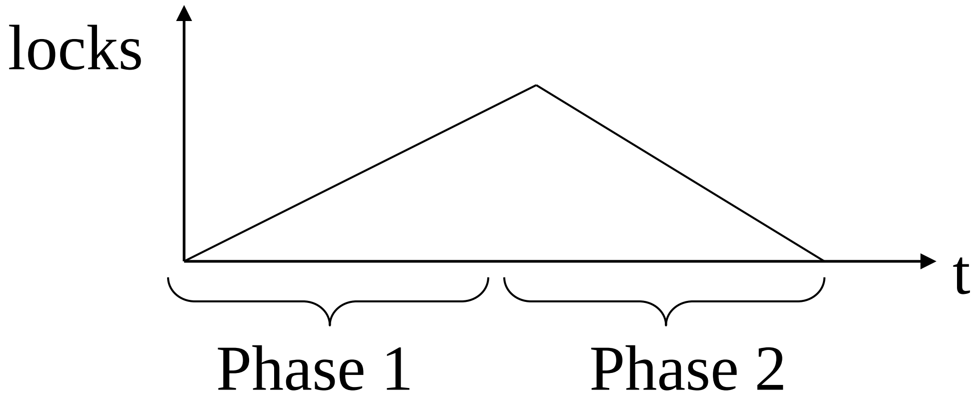
- Locks are associated with each data item
- A transaction must acquire a read (shared) or write (exclusive) lock on an item in order to read or write it
- A write lock on an item *conflicts* with all other locks on the item; a read lock conflicts with a write lock
- If *T1* requests a lock on *x* and *T2* holds a conflicting lock on *x*, *T1* must wait

Lock based CCT: 2 PL protocol

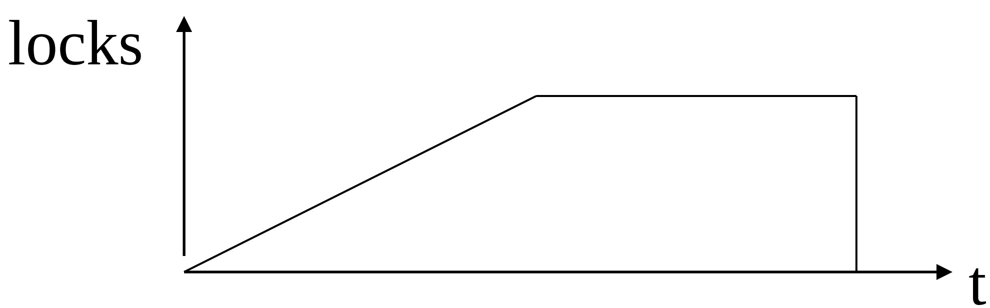
- Two-phase Locking (2PL): Ti follows 2PL if
  - all locking ops precede the first unlock op in T
- Two Phases relate to the text/execution of T
  - Phase 1: Expand / Grow the set of permissions (Locks)
  - Phase 2: Shrink - Release the Locks
- Variation of 2PL:
  - To get Strict Schedules (see pp757-759, Chap 21.4.2), Use Strict 2PL
  - Strict 2PL if T commits or aborts before any unlock() operations.
  - Strict 2PL does not avoid deadlocks.

Lock Release

Two-Phase locking: All locks are acquired before any lock is released



Strict: Transaction holds all locks until completion



Correctness of Strict Two-Phase Locking

- Intuition: Active transactions cannot have executed operations that do not commute (since locks required for non-commutative operations conflict)
- Hence, a schedule produced by a two-phase locking concurrency control is serializable since operations of concurrent transactions can always be reordered to produce a serial schedule

Non-Strict Concurrency Controls

- Non-strict controls: locks can be released before completion
- Problem: $w_1(x)\ u_1(x)\ r_2(x)\ w_2(y)\ commit_2\ \ abort_1$
  - Although $abort_1$ rolls *x* back, the new value of *y* may have been affected
  - *T1* has an effect even though it is aborted
  - Hence, atomicity is violated

Deadlock and Livelock Problems

- Livelocks or Deadlocks - possible with 2PL
- When a transaction can hold locks and request another lock (*e.g.*, in two-phase locking), a cycle of waiting transactions can result:
  - r_lock1(Y), r1(Y), r_lock2(X), r2(X), w_lock1(X), w_lock2(Y)
  - pp 786 Fig 22.5
- A transaction in the cycle must be aborted by DBMS (since transactions will wait forever)
- DBMS uses deadlock detection algorithms or timeout to deal with this problem

Lock based CCT: Solving Deadlock Problems

- Priority based on age (timestamp of first submission)
  - E.g. TS(T1) < TS(T2)
- Solution 1: Prevent Deadlocks
  - Conservative 2PL if T acquires all locks before T starts execution.
  - No deadlocks, if all locks requested together (no hold & wait).
- Sol. 2: Detect and Resolve Deadlocks
  - Wait-for graph (nodes = Ts, edge = wait-for dependency)
  - cycle in WFG = > deadlock
  - Break deadlock by aborting Ti to break cycles in WFG
- Sol. 3: Timeouts: Ti aborts if all locks are not granted within timeout!

  wait-die (new time-stamp) or wait-wound (no change in time-stamp)

CCT 2: Timestamp Ordering(pp594 18.2)

- IDEA: Make schedule eqv. to serial schedule
  - defined by the ordering of time-stamps (transX start time) of transactions
  - TS(T): timestamp of transX T
  - No deadlock
- Thus, Abort Ti operate on data-item X, such that
  - Ti reads /writes X with write_TS(X) > TS(Ti)
  - OR Ti writes X with read_TS(X) > TS(Ti)
  - WHERE
  - ...TS(Ti) = start time of a transaction Ti
  - ...read_TS(X) = largest TS(Ti) for any Ti that has already read X.
  - ...write_TS(X) = largest TS(Ti), any Ti has already written X.
- Timestamp Ordering ensures serializability
  - Note: Potential conflict detected before operation and avoided.
  - Conflict eqv. schedule = order Ti in S by their timestamps.
- Comparison of TSO, 2PL, CGS
  - 2PL/TSO limit concurrency: rules out some serializable schedules
  - 2PL and TSO produce different subsets of CGS

Other CCTs: Versions

- *18.3 Multiversion CCT*
  - Uses view serializability, not conflict serializability
  - Wi(X) writes a new version of X, old version of X remains
  - Some conflicting ri(X) can read older versions
  - Can be based on timestamps or 2PL
  - Oracle: timestamped-versions for read and 2PL for write

Other CCTs: Optimistic

- *22.4 Optimistic Validation Based CCT*
  - Idea: do all the checking at one time (commit time)
  - ...No side-effect to database before validation
  - Three phases: read + compute, validate, write
  - Abort Ti if conflict with committed / validated Xactions
  - Wins if there is small degree of conflicts across concurrent Ts
  - Too many aborts if too many conflicts