

CMSC 724: Concurrency

Amol Deshpande

University of Maryland, College Park

March 1, 2007

*Adapted from Joe Hellerstein's Notes (<http://redbook.cs.berkeley.edu/redbook3/lec4.html>)

Transactions and ACID

- Transaction: A sequence of actions bracketed by begin and end statements.
- ACID:
 - **Atomicity**: The entire transaction or nothing.
 - **Consistency**: Each transaction take DB from one consistent state to another.
 - **Isolation**: Events within a transaction invisible to other transactions.
 - **Durability**: Once committed, the results of a transaction preserved in spite of failures.
- C & I: Concurrency control; A & D: Recovery mechanisms.

Concepts

- Schedule, serial schedule, serializability. . .
- Recoverability: If a transaction is aborted, no other transactions should be affected
- Locking: To ensure transactions don't interfere
 - Share lock ("Read locks")
 - Exclusive lock ("Write locks")
- A "well-formed" transaction takes locks on the items it reads or updates.
- Two-phase locking:
 - Growing phase: Take locks, but don't release
 - Shrinking phase: Release locks, but don't take
- Well-formed and two-phase guarantees serializability.

Granularity of Locks

- Tradeoff: Fine granularity → high concurrency, but high overhead (vs coarse granularity)
- Granularities possible: DB, Areas, Files, Pages, Tuples (records), Fields of tuples
- Arrange in a hierarchy
 - Can be a DAG (could also be dynamically changing)
- New types of locks: Intention Shared (IS), Intention Exclusive (IX), Share and Intention Exclusive (SIX)
- Protocol:
 - Get at least intention locks on all ancestors
 - For DAGs: To get an X lock, must lock all paths from root to the node; for S locks, locking one path is enough.

Granularity of Locks

- Lock tables: Typically a in-memory hash table
- Lock/unlock themselves must be atomic instructions (why ?)
 - E.g. using semaphores or mutexes
 - Heavyweight: Several hundred instructions per call
- Lock conversion
- Starvation
- Deadlocks
 - Detection mechanisms (cycles in the waits-for graphs)
 - Avoidance mechanisms (pre-emptive mechanisms)

Degrees of Consistency

- Anomalies that may occur with concurrent transactions
 - Lost updates: (1) T1 writes X; (2) T2 writes X; (3) T1 aborts and restores back to old value; (4) T2 update has been lost
 - Dirty read: Reading the uncommitted data of another transaction
 - Repeatable read: If a transaction repeats a read, it should read the same value
 - Phantom read:
 - T1 reads all items in range [X, Y]
 - T2 inserts Z in that range
 - T1 re-reads all items in range [X, Y], and finds a new value (“phantom”)

Degrees of Consistency

- How do we reason about these and avoid such inconsistencies ?
- Note: ANSI SQL Isolation levels are defined using the latter three concepts.
 - Confusing and ambiguous (See Berenson et al., SIGMOD 1995)

Degrees of Consistency

- Definition using locking
 - Degree 0: set short write locks on updated items
 - Degree 1: set long write locks on updated items
 - Degree 2: Degree 1 and set short read locks on read items
 - Degree 3: Degree 1 and set long read locks on read items
- Whats “long” ?
 - End of the transaction to be safe
 - For serializability, sufficient to hold till the shrinking phase
 - But that won't guarantee recoverability

Degrees of Consistency

- Definition using the dirty reads. . .
- Transaction T “sees” (obeys) degree X consistency if:
 - Degree 0: T does not overwrite dirty data of other transactions
 - Degree 1: Degree 0, and T does not commit any writes before EOT
 - Degree 2: Degree 1, and T does not read dirty data of other transactions
 - Degree 3: Degree 2, and other transactions do not dirty any data read by T before T completes
- Much criticism afterwards – too vague and ambiguous (Berenson et al, SIGMOD 1995).

Degrees of Consistency

- Definition using the dirty reads. . .
- Corresponds to the definition using locking protocols
 - Not exactly: EOT vs shrinking phase
 - E.g. Degree 2 guarantees repeatable reads (though doesn't avoid phantom reads).
- Each transaction can choose its own degree of consistency so long as all transactions are at least degree 0 (degree 1?).
 - May not be correct.
 - T1: Degree 0 (short write locks), T2: Degree 3 (long read locks)
 - $w1[X] \dots r2[X] \dots T1\text{-Abort}$
 - Allowed even though T2 reads dirty data.
- Recoverability: Requires system-wide degree 1.

ANSI/SQL Isolation Levels

- Phenomenon: As defined by Berenson et al, 1995
 - P0: garbage writes - $w1[X] \dots w2[X] \dots$ (C1 or A1)
 - (C1 = Commit T1, A1 = Abort T1)
 - P1: dirty reads - $w1[X] \dots r2[X] \dots$ (C1 or A1)
 - P2: non-repeatable read - $r1[X] \dots w2[X] \dots$ (C1 or A1)
 - P4: phantom read - $r1[\text{pred}] \dots w2[\text{Y in pred}] \dots$ (C1 or A1)
- Isolation levels and locking implementations:
 - Degree 0; No restrictions
 - Degree 1 = Locking READ UNCOMMITTED – No P0
 - Degree 2 = Locking READ COMMITTED – No P0, P1
 - Locking REPEATABLE READ – No P0, P1, P2
 - Degree 3 = Locking SERIALIZABLE – No P0, P1, P2, P3

ANSI/SQL Isolation Levels

- Still locking-based definitions in disguise (Adya et al, ICDE 2000)
 - Degree 0: Short write locks
 - Locking READ UNCOMMITTED : Long write locks
 - Locking READ COMMITTED : Long write locks, short read locks
 - Locking REPEATABLE READ : Long write locks, long read locks
 - Locking SERIALIZABLE : Long write locks, long read predicate locks
- Doesn't allow some correct histories.
 - H1: w1[X] ... r2[X] ... COMMIT T1
 - This is okay if T2 is serialized after T1.
 - Locking-based implementations (assuming Degree 3 for both T1 and T2) disallow this.

Other Isolation Levels

- Snapshot isolation (called `SERIALIZABLE` in `ORACLE`)
 - Each transaction gets a timestamp
 - “First committer wins”: At commit time, if T1 and T2 have a WW conflict, and T1 commits, T2 is aborted.
 - Implementation: Archive old data and if a transaction asks for a data item, get the data from its start time.
 - A “multi-version” concurrency scheme.

Other Isolation Levels

- Snapshot isolation (SERIALIZABLE in ORACLE)
 - Not truly serializable
 - T1: $r(a_0)$, $r(b_0)$, $w(a_1 = a_0 - 100)$, C1
 - T2: $r(a_0)$, $r(b_0)$, $w(b_2 = b_0 - 100)$, C2
 - A is checking, B is savings (Constraint: $A + B > \$0$)
 - Initially: $A = 70$, $B = 70$
 - Each transaction checks for it, but constraint still not satisfied at the end
 - “Multi-object constraints”
 - Still commonly used (more concurrency).
- Cursor Stability (between Degree 1 and Degree 2)
 - Read-lock on a row as long as cursor is pointing to it

Optimistic Concurrency Control

- Simple idea: optimize case where conflict is rare.
 - Think cvs, svn etc. . .
- Basic idea: all transactions consist of three phases:
 - **Read:** All writes are to private storage (shadow copies).
 - **Validation:** Make sure no conflicts have occurred.
 - **Write:** If Validation successful, make writes public. (If not, abort!)
- Better in large-scale, wide-area distributed systems.
- When useful ?
 - All transactions are readers.
 - Lots of transactions, each accessing/modifying only a small amount of data, large total amount of data.
 - Fraction of transaction execution in which conflicts “really take place” is small compared to total pathlength.

The Validation Phase

- Goal: guarantee that only serializable schedules result.
- Technique: find an equivalent serializable schedule.
 - Assign each transaction a TN during execution.
 - Ensure that if you run transactions in order induced by “<” on TNs, you get an equivalent serial schedule.
- Suppose $TN(T_i) < TN(T_j)$. IF:
 - T_i completes its write phase before T_j starts its read phase.
 - $WS(T_i) \cap RS(T_j) = \phi$ and T_i completes its write phase before T_j starts its write phase.
 - $WS(T_i) \cap RS(T_j) = \phi$ and $WS(T_i) \cap WS(T_j) = \phi$ and T_i completes its read phase before T_j completes its read phase.
- THEN serializable (allow T_j writes).

Optimistic Concurrency Control: Details

- Maintain create, read, write sets
- Critical sections:
 - **tbegin:** record the current TN (called *start-tn*)
 - **tend:**
 - Validate against all transactions between *start-tn* and current transaction number.
 - If validated, assign the next TN to this transaction
 - If not validated, abort
 - Only one transaction can be in a critical section at any time
 - Read queries don't need to be assigned TNs

Optimistic Concurrency Control

- Need to store the read and write sets
 - Could become very large with long transactions
 - Large transactions also result in starvation
 - After enough aborts, lock the whole database (hopefully doesn't happen often)
- Write Phase:
 - May be too large to be done atomically inside the critical section
 - Also in case of parallel processors, critical sections may become bottlenecks
 - Split the write phase (move large parts outside critical section)
 - Allow interleaving of writes (with additional checks)

Locking vs Optimistic

- Many studies comparing the two
- Agrawal/Carey/Livny, ACM TODS 1987 (in redbook)
 - Very detailed performance study comparing blocking, optimistic and immediate restart (when locked, restart)
 - Models different kinds of scenarios (how many resources, processors etc). . .
 - Blocking usually wins over the other two
 - Optimistic better with low resource utilization, large think times etc. . .
- Most systems use locking
- Other issues:
 - Escrow transactions
 - Concurrency in indexes