Transaction Management

Introduction to Transaction

- Concurrent execution of user programs is essential for good DBMS performance.
- A transaction is the DBMS's abstract view of a user program: a sequence of READs and WRITEs.
- DBMS ensures ACID of transactions:
  - Atomicity: either all actions are carried or none are.
  - Consistency: when a user's transaction runs to completion by itself against a 'consistent' database instance, it will leave the database in a 'consistent' state.
  - Isolation: users can understand a transaction without considering the effect of other concurrently executing transactions.
  - Durability: once a transaction has been successfully completed, its effects should persist even if the system crashes before all its changes are reflected on disk.

DBMS Concurrency

- Users submit transactions, and can think of each transaction as executing by itself.
  - Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
  - Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
    - DBMS will enforce some ICs, depending on the ICs declared in CREATE TABLE statements.
    - Beyond this, the DBMS does not really understand the semantics of the data.
Scheduling Transactions

- A transaction is a series of actions including READs / WRITEs of database objects, ABORTs and COMMITs.
- A schedule is a list of actions from a set of transactions, and the order in which two actions of a transaction appear in a schedule must be the same as the order in which they appear in the transaction.
- Serial schedule: Schedule that does not interleave the actions of different transactions.

Scheduling Transactions (Cont)

- Equivalent schedules: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
- Serializable schedule: A schedule that is equivalent to some serial execution of the transactions.
- If each transaction preserves consistency, every serializable schedule preserves consistency.

Anomalies Due To Interleaved Execution

- Two actions on the same data object conflict if at least one of them is a WRITE.
  - write-read (WR) conflict: T2 could read a DB object that has been modified by T1, which has not yet committed.
  - read-write (RW) conflict: T2 could change the value of an object that has been read by T1, while T1 is still in progress.
  - write-write (WW) conflict: T2 could overwrite the value of an object which has already been modified by T1, while T1 is still in progress.
Schedules Involving ABORTs

- A **serializable schedule** is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over the set of committed transactions.
- In a **recoverable schedule**, transactions commit only after all transactions whose changes they read commit. (Avoid cascading aborts)

Lock-based Concurrency Control

- A **lock** is a small bookkeeping object associated with a database object.
- A **locking protocol** is a set of rules to be followed by each transaction to ensure that, even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions in some serial order.

Strict Two-Phase Locking Protocol

**Strict 2PL Protocol:**

- Each transaction must obtain a $S$ (shared) lock on object before reading, and an $X$ (exclusive) lock on object before writing.
- All locks held by a transaction are released when the transaction completes.
- If a transaction holds an $X$ lock on an object, no other transactions can get a lock ($S$ or $X$) on that object.

Strict 2PL allows only serializable schedules.
Aborting a Transaction

- If a transaction $T_i$ is aborted, all its actions have to be undone. Not only that, if $T_j$ reads an object last written by $T_i$, $T_j$ must be aborted as well.
- Most systems avoid such cascading aborts by releasing a transaction’s locks only at commit time.
- If $T_i$ writes an object, $T_j$ can read this only after $T_i$ commits.
- In order to undo the actions of an aborted transaction, the DBMS maintains a log in which every write is recorded. This mechanism is also used to recover from system crashes: all active transactions at the time of the crash are aborted when the system comes back up.

Crash Recovery

- A recovery manager of a DBMS is responsible for ensuring transaction atomicity and durability.
- Can the changes made to an object in the buffer pool by a transaction $T$ be written to disk before $T$ commits? (steal?)
- When a transaction commits, must all the changes it has made to objects in the buffer pool are immediately forced to disk? (force?)
- Most systems use a steal, no-force approach.

Concurrency Control Techniques

Instructor: Dr. Yingshu Li
yli@cs.gsu.edu
Conflict Serializable Schedules

- Two schedules are conflict equivalent if:
  - Involve the same actions of the same transactions
  - Every pair of conflicting actions is ordered the same way
- A schedule is conflict serializable if it is conflict equivalent to some serial schedule.
  - Every conflict serializable schedule is serializable
  - Some serializable schedules are not conflict serializable.

Dependency Graph

- A dependency graph (serializable graph) captures all the potential conflicts between the transactions in a schedule. The dependency graph for a schedule $S$ contains:
  - A node for each committed transaction in $S$.
  - An arc from $T_i$ to $T_j$ if an action of $T_i$ precedes and conflicts with one of $T_j$'s actions.

Theorems

- A schedule is conflict serializable if and only if its dependency graph is acyclic.
- Strict 2PL protocol allows only conflict serializable schedules.
  - Each transaction must obtain a $S$ (shared) lock on object before reading, and a $X$ (exclusive) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes
  - If a transaction holds a $X$ lock on an object, no other transaction can get a lock ($S$ or $X$) on that object.
Two-Phase Locking

- **Two-Phase Locking (2PL) protocol**
  - Each transaction must obtain a S (shared) lock on object before reading, and an X (exclusive) lock on object before writing.
  - A transaction can not request additional locks once it releases any locks.
  - If a transaction holds an X lock on an object, no other transaction can get a lock (S or X) on that object.
- 2PL allows only conflict serializable schedules.

Strict 2PL vs. 2PL

- A schedule is **strict** if an object written by a transaction $T$ is not read or overwritten by another transaction until $T$ commits/aborts.
- **Strict 2PL** improves on 2PL by guaranteeing that every allowed schedule is strict in addition to being conflict serializable.

Lock Management

- The **lock manager** is in charge of the locks issued to transactions.
- Lock table entry:
  - Object ID
  - Number of transactions currently holding a lock on an object
  - Type of lock held (shared or exclusive)
  - Pointer to queue of locks requests
- Transaction table entry:
  - Transaction information
  - Pointer to a list of locks held by each transaction
- Locking and unlocking have to be atomic operations
- Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock.
Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
  - Deadlock detection
  - Deadlock prevention

Deadlock Detection

- Construct a *Waits-for* graph:
  - Nodes are transactions
  - There is an edge from $T_i$ to $T_j$ if $T_i$ is waiting for $T_j$ to release a lock
- Periodically check for cycles in the *waits-for* graph.

Deadlock Prevention

- Assign priorities based on *timestamps*.
  - The lower the timestamp, the higher the transaction’s priority.
- Assume $T_i$ wants a lock that $T_j$ holds. Two policies are possible:
  - **Wait-Die**: If $T_i$ has higher priority, $T_i$ waits for $T_j$; otherwise $T_j$ aborts
  - **Wound-wait**: If $T_i$ has higher priority, $T_j$ aborts; otherwise $T_i$ waits
- If a transaction re-starts, make sure it has its original timestamp.
Deadlock Prevention (Cont)

- Conservative 2PL:
  - A transaction obtains all the locks it will ever need when it begins, or blocks waiting for these locks to become available.
- Reduce the time that locks are held on average.
- Hard to decide exactly what locks are needed ahead of time
- Lead to setting more locks than needed.
- Higher overhead for setting locks.

Optimistic Concurrency Control

- Locking is a conservative approach in which conflicts are prevented. Disadvantages:
  - Lock management overhead.
  - Deadlock detection/resolution.
  - Lock contention for heavily used objects.
- If conflicts are rare, we might be able to gain concurrency by not locking, and instead checking for conflicts before transactions commit.

Kung-Robinson Model

- Transactions have three phases:
  - READ: transactions read from the database, but make changes to private copies of objects.
  - VALIDATION: Check for conflicts. If conflicts are possible, abort and restart the transaction.
  - WRITE: If no conflicts, make local copies of changes public.
Validation

- Test conditions that are sufficient to ensure that no conflict occurred.
- Each transaction is assigned a **timestamp** $TS(T_i)$.
- Each timestamp is assigned at end of READ phase, just before VALIDATION begins.
- $ReadSet(T_i)$: Set of objects read by $T_i$.
- $WriteSet(T_i)$: Set of objects modified by $T_i$.

Validation Conditions

- To validate $T_j$, must check to see that one of the conditions holds with respect to each committed transaction $T_i$ where $TS(T_i) < TS(T_j)$. (It ensures that $T_j$'s modifications are not visible to $T_i$).
  - $T_i$ completes all three phases before $T_j$ begins.
  - $T_i$ completes before $T_j$ begins its WRITE phase and $WriteSet(T_i) \cap ReadSet(T_j) = \emptyset$.
  - $T_i$ completes READ phase before $T_j$ does, $WriteSet(T_i) \cap ReadSet(T_j) = \emptyset$ and $WriteSet(T_i) \cap WriteSet(T_j) = \emptyset$.

Overheads in Optimistic Concurrency Control

- Must maintain Read/Write Sets per transaction.
- Must check for conflicts during validation, and must make validated writes "global".
  - **Critical section** can reduce concurrency.
  - Scheme for making writes global can reduce clustering of objects.
- Optimistic concurrency control restarts transactions that fail validation.
  - Work done so far is wasted; requires clean-up.
Optimistic 2PL

- **Optimistic 2PL:**
  - Set S locks as usual.
  - Make changes to private copies of objects.
  - Obtain all X locks at end of transaction, make writes global, then release all locks.
- In contrast to Optimistic CC as in Kung-Robinson, this scheme results in transactions being blocked, waiting for locks.
  - However, no validation phase, no restarts.

Timestamp-based Concurrence Control

- Give each object a read-timestamp (RTS) and a write-timestamp (WTS), give each transaction a timestamp (TS) when it begins:
  - If action $a_i$ of transaction $T_i$ conflicts with action $a_j$ of transaction $T_j$ and $TS(T_i) < TS(T_j)$, then $a_i$ must occur before $a_j$. Otherwise, restart violating transaction.

When $T$ Wants to Read $O$

- If $TS(T) > WTS(O)$:
  - Allow $T$ to read $O$.
  - Reset $RTS(O)$ to $max(RTS(O), TS(T))$
- If $TS(T) < WTS(O)$, this violates timestamp order of $T$ w.r.t. writer of $O$.
  - So, abort $T$ and restart it with a new, larger $TS$. (If restarted with same $TS$, $T$ will fail again! Contrast use of timestamps in 2PL for deadlock prevention.)
- Change to $RTS(O)$ on reads must be written to disk! This and restarts represent overheads.
When $T$ Wants to Write $O$

- If $TS(T) < RTS(O)$, this violates timestamp order of $T$ w.r.t. reader of $O$; abort and restart $T$.
- If $TS(T) < WTS(O)$, violates timestamp order of $T$ w.r.t. writer of $O$.
  - **Thomas Write Rule**: We can safely ignore such outdated writes; need not restart $T$! ($T$'s write is effectively followed by another write, with no intervening reads). Allows some serializable but non-conflict serializable schedules.
- Else, allow $T$ to write $O$. 