Transaction Management: Concurrency Control

Yanlei Diao
UMass Amherst

Outline

- Transaction management overview
- Serializability & recoverability
- Lock-based concurrency control
- Optimistic concurrency control
- Efficient B+tree locking

Concurrent User Programs

- Concurrent execution of user programs: good for performance.
  - When task 1 is doing I/O, run task 2 to utilize the CPU.
  - Improve average response time (average delay that a user task experiences)
  - Improve system throughput (number of user tasks processed in each time unit)

Transactions

- User programs may do many things on the data retrieved.
  - E.g., operations on Bob’s bank account.
  - E.g. transfer of money from account A to account B.
  - E.g., search for a ticket, think about it…, and buy it.
- But the DBMS is only concerned about what data is read from/written to the database.
- A transaction is DBMS’s abstract view of a user program, simply, a sequence of reads and writes.

Concurrency

- Many users submit xacts, but each user thinks of his as executing by itself.
  - DBMS interleaves reads and writes of xacts for concurrency.
- Consistency: each xact starts and ends with a consistent state (i.e., satisfying all integrity constraints).
  - E.g., if an IC states that all accounts must have a positive balance, no transaction can violate this rule.
- Isolation: execution of one xact appears isolated from others.
  - Nobody else can see the data in its intermediate state, e.g., account A being debited but B not being credited.
Recovery

- A transaction might commit after completing all its actions, or it could be aborted after executing some actions.
- Atomicity: either all actions of a xact are performed or none of them is (all-or-none).
  - DBMS logs all actions so that it can undo the actions of aborted xacts.
- Durability: once a user program has been notified of success, its effect will persist despite system failure.
  - DBMS logs all actions so that it can redo the actions of committed xacts.

Outline

- Transaction management overview
  - Serializability & recoverability
- Lock-based concurrency control
- Optimistic concurrency control
- Efficient B+tree locking

Example

- Consider two transactions:

  T1: BEGIN A=A+100, B=B-100 END
  T2: BEGIN A=1.06*A, B=1.06*B END

  - 1st xact transfers $100 from B’s account to A’s.
  - 2nd xact credits both accounts with a 6% interest payment.
  - No guarantee that T1 will execute before T2 or vice-versa, if both are submitted together.

  However, the net effect must be equivalent to these two transactions running serially in some order!

Example (Contd.)

- Consider a possible interleaving schedule:

  T1: A=A+100, B=B-100
  T2: A=1.06*A, B=1.06*B

  - This is OK. But what about:

    T1: A=A+100, B=B-100
    T2: A=1.06*A, B=1.06*B

  The DBMS’s view of the second schedule:

    T1: R(A), W(A), R(B), W(B)
    T2: R(A), W(A), R(B), W(B)

Scheduling Transactions

- Serial schedule: Schedule that does not interleave the actions of different transactions.
- Equivalent schedules: For any database state, the effect 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.

Serializability

- Serializability theory concerns the schedules of transactions that are not (explicitly) aborted.
- Given a set of such xacts, ideally want to allow any serializable schedule.
  - Recognizing any serializable schedule is highly complex, if possible.
- Instead, allow only a subset of serializable schedules that are easy to detect.
Conflict Serializability

- Two schedules are **conflict equivalent** if:
  - Involve the same actions of the same transactions.
  - Every pair of conflicting actions is ordered the same way.
- Schedule S is **conflict serializable** if S is conflict equivalent to some serial schedule.
- Given a set of xacts, conflict serializable schedules are a **subset** of serializable schedules.
  - There are serializable schedules that can’t be detected using conflict serializability.

Dependency Graph

- **Precedence graph**:
  - One node per Xact;
  - Edge from Xact Ti to Xact Tj if an action of Ti **precedes** and **conflicts with** one of Tj’s actions (**RW, WR, WW operations on the same object**).
- **Theorem**: Schedule is conflict serializable if and only if its precedence graph is acyclic.

Example

- **The schedule is not conflict serializable**:
  - The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.

Weaker Condition on Serializability

- Conflict serializability is **sufficient but not necessary** for serializability.

View Serializability

- Schedules S1 and S2 are **view equivalent** if:
  - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - If Ti writes final value of A in S1, then Ti also writes final value of A in S2

View Serializability (Contd.)

- A schedule is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is view serializable.
  - The converse is not true.
- Every view serializable schedule that is not conflict serializable contains a **blind write**.
Recoverability

- **Recoverability theory** concerns schedules that involve aborted transactions.

<table>
<thead>
<tr>
<th>T1: R(A), W(A)</th>
<th>Abort</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2: R(A), W(A)</td>
<td>Commit</td>
</tr>
</tbody>
</table>

Unrecoverable!

- A schedule S is **recoverable** if each xact commits only after all xacts from which it read have committed.

Recoverability (Contd.)

<table>
<thead>
<tr>
<th>T1: R(A), W(A)</th>
<th>Abort</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2: R(A), W(A)</td>
<td>Commit</td>
</tr>
</tbody>
</table>

Recoverable, but with cascading aborts.

- S **avoids cascading rollback** if each xact may **read** only those values written by committed xacts.

Recoverability (Contd.)

<table>
<thead>
<tr>
<th>T1: R(A), W(A)</th>
<th>Abort</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2: R(A), W(A)</td>
<td>Commit</td>
</tr>
</tbody>
</table>

Recoverable, no cascading aborts, but update of A by T2 will be lost!

- S is **strict** if each xact may **read and write** only objects previously written by committed xacts.
  - No cascading aborts.
  - Actions of aborted xacts can be simply undone by restoring the original values of modified objects.

Venn Diagram for Schedules

- **Strict 2PL**

- **Strict Two-Phase Locking (Strict 2PL) Protocol:**
  - Each Xact must obtain a **S (shared)** lock on object before reading, an X (exclusive) lock on object before writing.
  - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
  - All locks held by a transaction are released when the transaction completes.
Strict 2PL (contd.)

- **Theorem:** Strict 2PL allows only schedules whose precedence graph is acyclic.
  - Strict 2PL only allows *conflict serializable* schedules.
- Strict 2PL is strict with respect to recoverability.
  - Strict 2PL is recoverable without anomalies related to aborted transactions.
  - Hence, it simplifies transaction aborts.

Nonstrict 2PL (contd.)

- **Theorem:** Nonstrict 2PL ensures acyclicity of precedence graph.
  - Nonstrict 2PL only allows *conflict serializable* schedules.
  - An equivalent serial schedule is given by the order of xacts entering their *shrinking phase*.
- Nonstrict 2PL is recoverable but *not strict*!
  - Involves complex abort processing.
  - But allows xacts to go through more quickly.

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

- Create a *waits-for graph*:
  - Nodes are Xacts.
  - There is an edge from Xact Ti to Xact Tj if Ti is *waiting for* Tj to release a lock.
  - Note the difference from the precedence graph for conflict serializability.
- Periodically check for cycles, indicating deadlocks, in the waits-for graph.
  - Resolve a deadlock by aborting a transaction on the cycle and releasing all its locks.
**Deadlock Prevention**

- Assign priorities based on timestamps.
  - The older the timestamp, the higher the xact’s priority.
- **Wait-Die**: Ti wants a lock that Tj holds. If Ti has higher priority, Ti waits for Tj; otherwise Ti aborts.
  - Lower priority xacts can never wait.
- **Wound-wait**: Ti wants a lock that Tj holds. If Ti has higher priority, Tj aborts; otherwise Ti waits.
  - Higher priority xacts never wait.
- If a transaction re-starts, make sure it has its original timestamp so its priority increases.

**The Phantom Problem**

- If we consider insertion and update of records in a DB, even Strict 2PL won’t assure serializability:
  - T1 locks all pages containing sailor records with rating = 1, and finds oldest sailor, say, age = 71.
  - T2 next inserts a new sailor: rating = 1, age = 96.
  - T1 now reads reads the oldest sailor again; now, age = 96 appears as a phantom!
- **Problem**: T1 assumes that it has locked all sailor records with rating = 1.
  - This is only true if nobody adds/updates records while T1 is running!

**The Phantom Problem (Contd.)**

- Phantoms can occur with both inserts and updates.
  - E.g., update a sailor’s rating from 3 to 1.
- Need a mechanism to enforce the assumption that T1 indeed holds locks on all records satisfying a condition.
  - Index locking: used in practice
  - Predicate locking: impractical, not discussed in this class

**Index Locking**

- If there is an index on the rating field, T1 should lock the index page containing data entries with rating = 1.
  - If there are no data entries with rating = 1, T1 must lock the index page where such a data entry would be, if it existed!
- If there is no suitable index, T1 must lock the entire file/table to prevent new pages from being added.
  - so that no new records with rating = 1 can be added.

**Lock Management**

- **Lock manager** handles the lock and unlock requests
- **Transaction table**: xact id → <locks held in xact>
- **Lock table**: object id → lock table entry
  - Object can be a page, a record, etc.
  - Lock table entry:
    - Number of transactions currently holding a lock
    - Type of lock held (shared or exclusive)
    - Queue of lock requests

**Lock and Unlock**

- When an xact requests a lock on an object O:
  - If an S lock is requested, O is not currently locked in the X mode and the queue of requests is empty, then...
  - If an X lock is requested, no xact currently holds a lock on O, then...
  - Otherwise, ...
- When an xact aborts or commits:
  - it releases all its locks, and
  - when the lock on an object O’ is released, its lock table entry is updated and the request at the head of the queue is answered.
Other Lock Management Issues

- Lock and unlock have to be atomic operations.
  - Important when several instances of the lock manager code can execute concurrently.
  - Implemented using an OS synchronization mechanism such as a semaphore.
- Lock upgrade: An xact that holds a shared lock can be upgraded to hold an exclusive lock.
  - Important for avoiding deadlocks.
- Convoy: An xact that holds a heavily used lock can be suspended by the OS. Every other xact requesting the lock is queued and the queue can grow very long.
  - Drawback of building a DBMS on top of a general-purpose OS with preemptive scheduling.

Outline

- Transaction management overview
- Serializability & recoverability
- Lock-based concurrency control
  - Optimistic concurrency control
- Efficient B+tree locking

Optimistic CC (Kung-Robinson)

- Locking is a conservative approach to preventing conflicts:
  - Lock management overhead.
  - Deadlock detection/resolution.
  - Lock contention for heavily used objects.
  - Phantom problem.

- Idea: If conflicts are rare, we gain concurrency if we do not lock, and instead check for conflicts before Xacts commit.

Kung-Robinson Model

- Xacts have three phases:
  - READ: Xacts read from the database, but make changes to local (private) copies of objects.
  - VALIDATE: Check for conflicts. If there is a conflict, abort (clear the local copy & restart).
  - WRITE: Make local copies of changes public.

Validation

- Goal: guarantee that only serializable schedules result.
- Technique: find an equivalent serializable schedule
  - Assign an Xact id (timestamp) to each xact at the beginning of the validation phase.
  - Check if the timestamp-ordering of xacts is an equivalent serial order.
- Three test conditions ensure an equivalent serializable schedule.
  - ReadSet(Ti): objects read by Xact Ti
  - WriteSet(Ti): objects modified by Ti

Test 1

- For all i and j such that Ti < Tj, check that Ti completes before Tj begins.
If Test 1 fails, try Test 2…

- For all \(i\) and \(j\) such that \(Ti < Tj\), check that:
  - \(Ti\) completes before \(Tj\) begins its Write phase, and
  - \(\text{WriteSet}(Ti) \cap \text{ReadSet}(Tj)\) is empty.

Does \(Tj\) read dirty data? Does \(Tj\) overwrite \(Ti\)'s writes?

- Check correctness: all three types of conflicts, \(W-R\), \(R-W\), \(W-W\), if present, go one way only.

If Test 2 fails, try Test 3

- For all \(i\) and \(j\) such that \(Ti < Tj\), check that:
  - \(Ti\) completes Read phase before \(Tj\) does, and
  - \(\text{WriteSet}(Ti) \cap \text{ReadSet}(Tj)\) is empty, and
  - \(\text{WriteSet}(Ti) \cap \text{WriteSet}(Tj)\) is empty.

Does \(Tj\) read dirty data? Does \(Tj\) overwrite \(Ti\)'s writes?

- Why is it correct?

Comments on Optimistic CC

- Compared to Locking
  - Optimistic CC assumes no conflicts first, only fixes problems when conflicts appear, by restarting xacts.
  - Locking (pessimistic): conflicts are prevented in advance, by blocking from (potentially) nonserializable actions.

- Works well for some workloads:
  - All xacts are readers.
  - Low interference, e.g. large amount of data, each xact accessing a small (likely non-overlapping) amount of data.

- Deadlock free, but may have starvation.

- No phantom problem!

Overheads in Optimistic CC

- Record read/write activity in ReadSet/WriteSet per Xact.
  - Must create and destroy these sets as needed.

- Check for conflicts during validation
  - Code for validation is in a critical section, and critical section can reduce concurrency.

- Make validated writes "global".
  - Scheme for making writes global can reduce clustering of objects. Sequential I/O is unlikely later.

- Restart Xacts that fail validation.
  - Work done so far is wasted; requires clean-up.
  - Starvation may occur.