**Transaction Management Overview**

Chapter 16

---

**Why Is This Important?**

- How can we perform multiple DB operations as one atomic unit?
  - Example: insert new dorm building
    - First insert building into DormBuilding: rejected, because no rooms registered for it in RoomContain
    - First insert rooms into RoomContain: rejected, because building does not exist yet in DormBuilding
- How does the DBMS enforce correct query execution when multiple queries and updates run in parallel?
- How can we improve performance by weakening consistency guarantees?

---

**Transactions**

- Concurrent execution of user programs is essential for good DBMS performance.
  - While some request is waiting for I/O, CPU can work on another one.
- A user’s program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database.
- A transaction is the DBMS’s abstract view of a user program: a sequence of reads and writes.

---

**Concurrency in a DBMS**

- 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 all specified constraints.
  - Beyond this, the DBMS does not really understand the semantics of the data. (E.g., it does not understand how the interest on a bank account is computed.)
- Issues: Effect of interleaving transactions and crashes.

---

**The ACID Properties**

- Atomicity: Either all or none of the transaction’s actions are executed
  - Even when a crash occurs mid-way
- Consistency: Transaction run by itself must preserve consistency of the database
  - User’s responsibility
- Isolation: Transaction semantics do not depend on other concurrently executed transactions
- Durability: Effects of successfully committed transactions should persist, even when crashes occur

---

**Example**

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

- T1 transfers $100 from B’s account to A’s account.
- T2 credits both accounts with a 6% interest payment.
- There is 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):
  
<table>
<thead>
<tr>
<th>Schedule 1</th>
<th>Schedule 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: A=A+100, B=B-100</td>
<td>T2: A=1.06<em>A, B=1.06</em>B</td>
</tr>
<tr>
<td>T1: A=A+100, B=B-100</td>
<td>T2: A=1.06<em>A, B=1.06</em>B</td>
</tr>
</tbody>
</table>

  This is OK. But what about:

<table>
<thead>
<tr>
<th>Schedule 1</th>
<th>Schedule 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: A=A+100, B=B-100</td>
<td>T2: A=1.06<em>A, B=1.06</em>B</td>
</tr>
<tr>
<td>T1: A=A+100, B=B-100</td>
<td>T2: A=1.06<em>A, B=1.06</em>B</td>
</tr>
</tbody>
</table>

  The DBMS's view of the second schedule:

<table>
<thead>
<tr>
<th>Schedule 1</th>
<th>Schedule 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: R(A), W(A), R(B), W(B)</td>
<td>T2: R(A), W(A), R(B), W(B)</td>
</tr>
</tbody>
</table>

Scheduling Transactions

- Serial schedule: Schedule that does not interleave the actions of different transactions.
  - Easy for programmer, easy to achieve consistency
  - Bad for performance

- Equivalent schedules: For any database state, the effect (on the 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.
  - Retains advantages of serial schedule, but addresses performance issue
  - Note: If each transaction preserves consistency, every serializable schedule preserves consistency.

Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, “dirty reads”)
  - Example: T1(A=A-100), T2(A=1.06A), T2(B=1.06B), C(T2), T1(B=B+100)
  - T2 reads value A written by T1 before T1 completed its changes
  - Notice: If T1 later aborts, T2 worked with invalid data

More Anomalies

- Unrepeatable Reads (RW Conflicts)
  - T1 sees two different values of A, even though it did not change A between the reads
  - Example: online bookstore
    - Only one copy of a book left
    - Both T1 and T2 see that 1 copy is left, then try to order
    - T1 gets an error message when trying to order
    - Could not have happened with serial execution

Even More Anomalies

- Overwriting Uncommitted Data (WW Conflicts)
  - T1’s B and T2’s A persist, which would not happen with any serial execution
  - Example: 2 people with same salary
    - T1 sets both salaries to 2000, T2 sets both to 1000
    - Above schedule results in A=1000, B=2000, which is inconsistent

Aborted Transactions

- All actions of aborted transactions have to be undone
  - Dirty read can result in unrecoverable schedule
    - T1 writes A, then T2 reads A and makes modifications based on A’s value
    - T2 commits, and later T1 is aborted
    - T2 worked with invalid data and hence has to be aborted as well; but T2 already committed...

- Recoverable schedule: cannot allow T2 to commit until T1 has committed
  - Can still lead to cascading aborts
Preventing Anomalies through Locking

- DBMS can support concurrent transactions while preventing anomalies by using a locking protocol.
- If a transaction wants to read an object, it first requests a shared lock (S-lock) on the object.
- If a transaction wants to modify an object, it first requests an exclusive lock (X-lock) on the object.
- Multiple transactions can hold a shared lock on an object.
- At most one transaction can hold an exclusive lock on an object.

Lock-Based Concurrency Control

- Strict Two-phase Locking (Strict 2PL) Protocol:
  - Each Xact must obtain the appropriate lock before accessing an object.
  - All locks held by a transaction are released when the transaction is completed.
  - All this happens automatically inside the DBMS.
- Strict 2PL allows only serializable schedules.
  - Prevents all the anomalies shown earlier.

The Phantom Problem

- Assume initially the youngest sailor is 20 years old.
- T1 contains this query twice.
  - SELECT rating, MIN(age) FROM Sailors.
- T2 inserts a new sailor with age 18.
- Consider the following schedule:
  - T1 runs query, T2 inserts new sailor, T1 runs query again.
  - T1 sees two different results! Unrepeatable read.
- Would Strict 2PL prevent this?
  - Assume T1 acquires Shared lock on each existing sailor tuple.
  - T2 inserts a new tuple, which is not locked by T1.
  - T2 releases its Exclusive lock on the new sailor before T1 reads Sailors again.
- What went wrong?

What Should We Lock?

- T1 cannot lock a tuple that T2 will insert.
- ...but T1 could lock the entire Sailors table.
  - Now T2 cannot insert anything until T1 completed.
- What if T1 computed a slightly different query:
  - SELECT MIN(age) FROM Sailors WHERE rating = 8.
- Now locking the entire Sailors table seems excessive, because inserting a new sailor with rating <> 8 would not create a problem.
  - T1 can lock the predicate [rating = 8] on Sailors.
- General challenge: DBMS needs to choose appropriate granularity for locking.

Deadlocks

- Assume T1 and T2 both want to read and write objects A and B.
  - T1 acquires X-lock on A; T2 acquires X-lock on B.
  - Now T1 wants to update B, but has to wait for T2 to release its lock on B.
  - But T2 wants to read A and also waits for T1 to release its lock on A.
  - Strict 2PL does not allow either to release its locks before the transaction completed. Deadlock!
- DBMS can detect this:
  - Automatically breaks deadlock by aborting one of the involved transactions.
  - Tricky to choose which one to abort: work performed is lost.

Performance of Locking

- Locks force transactions to wait.
- Abort and restart due to deadlock wastes the work done by the aborted transaction.
  - In practice, deadlocks are rare, e.g., due to lock downgrades approach.
- Waiting for locks becomes bigger problem as more transactions execute concurrently.
  - Allowing more concurrent transactions initially increases throughput, but at some point leads to thrashing.
  - Need to limit max number of concurrent transactions to prevent thrashing.
  - Minimize lock contention by reducing the time a Xact holds locks and by avoiding hotspots (objects frequently accessed).
**Controlling Locking Overhead**

- Declaring Xact as “READ ONLY” increases concurrency.
- **Isolation level**: trade off concurrency against exposure of Xact to other Xact’s uncommitted changes.

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Dirty Read</th>
<th>Unrepeatable Read</th>
<th>Phantom</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>Maybe</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>No</td>
<td>Maybe</td>
<td></td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>No</td>
<td>No</td>
<td>Maybe</td>
</tr>
<tr>
<td>Serializable</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Locking vs. Isolation Levels**

- **Serializable**: obtains locks on (sets of) accessed objects and holds them until the end.
- **Repeatable Read**: same locks as for serializable Xact, but does not lock sets of objects at higher level.
- **Read Committed**: obtains X-locks before writing and holds them until the end; obtains S-locks before reading, but releases them immediately after reading.
- **Read Uncommitted**: does not obtain S-locks for reading; not allowed to perform any writes.
  - Does not request any locks ever.

**Summary**

- Concurrency control is one of the most important functions provided by a DBMS.
- Users need not worry about concurrency.
  - System automatically inserts lock/unlock requests and can schedule actions of different Xacts in such a way as to ensure that the resulting execution is equivalent to executing the Xacts one after the other in some order.
- DBMS automatically undoes the actions of aborted transactions.
  - Consistent state: Only the effects of committed Xacts seen.