Intro to Transaction Management

CMPSCI 645
May 3, 2006

Gerome Miklau

Slide content adapted from Ramakrishnan & Gehrke, Zack Ives
Concurrency Control

- Concurrent execution of user programs is essential for good DBMS performance
- We must also cope with partial operations
- The transaction is the foundation for:
  - Concurrent execution
  - Recovery from system failure, incomplete ops
What is a Transaction?

- A **transaction** is the DBMS’s abstract view of a user program: a sequence of reads and writes.
The ACID Properties

- Particularly important: ensuring ACID properties
  - Atomicity
  - Consistency
  - Isolation
  - Durability
Atomicity

- A very important property guaranteed by the DBMS for all transactions is that they are **atomic**.
  - That is, a user can think of a Xact as always executing all its actions in one step, or not executing any actions at all.
  - DBMS logs all actions so that it can undo the actions of aborted transactions.
- If it succeeds, the effects of write operations persist (**commit**);
- If it fails, no effects of write operations persist (**abort**)
Consistency

- 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. (e.g., it does not understand how the interest on a bank account is computed).
Isolation

- Many concurrent transactions are running at one time.
- Each transaction should be isolated from the effects of other transactions.
- The net effect of concurrently running \{T1 and T2 and T3\} is identical to some serial order
  - No guarantee which serial order
Durability

- If transaction completes, its effects will persist in the database.
- In particular, if the system crashes before effects are written to disk, they will be redone.
- Recovery manager is responsible for this.
The ACID Properties

- Particularly important: ensuring ACID properties
  - Atomicity: each operation looks atomic to the user
  - Consistency: each operation in isolation keeps the database in a consistent state (this is the responsibility of the user)
  - Isolation: should be able to understand what’s going on by considering each separate transaction independently
  - Durability: updates stay in the DBMS!!!
Example

- Consider two transactions (Xacts):
  - The first transaction is transferring $100 from B’s account to A’s account.
  - The second is crediting 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.
Interleaving operations

<table>
<thead>
<tr>
<th>T1: Transfer</th>
<th>T2: Interest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin</td>
<td>Begin</td>
</tr>
<tr>
<td>A=A+100</td>
<td>A=1.06*A</td>
</tr>
<tr>
<td>B=B-100</td>
<td>B=1.06*B</td>
</tr>
<tr>
<td>End</td>
<td>End</td>
</tr>
</tbody>
</table>
Interleaving operations

<table>
<thead>
<tr>
<th>T1: Transfer</th>
<th>T2: Interest</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=A+100</td>
<td>A=1.06*A</td>
</tr>
<tr>
<td>B=B-100</td>
<td>B=1.06*B</td>
</tr>
</tbody>
</table>

Is this interleaving okay?
Interleaving operations

<table>
<thead>
<tr>
<th>T1: Transfer</th>
<th>T2: Interest</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A = A + 100$</td>
<td>$A = 1.06 \times A$</td>
</tr>
<tr>
<td>$B = B - 100$</td>
<td>$B = 1.06 \times B$</td>
</tr>
</tbody>
</table>

How about this interleaving?
Scheduling Transactions

- A transaction is seen by DBMS as sequence of reads and writes
  - read of object O denoted R(O)
  - write of object O denoted W(O)
  - must end with Abort or Commit

- A schedule of a set of transactions is a list of all actions where order of two actions from any transaction must match order in that transaction.
### A schedule

<table>
<thead>
<tr>
<th>T1: Transfer</th>
<th>T2: Interest</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = A + 100</td>
<td>A = 1.06*A</td>
</tr>
<tr>
<td>B = B - 100</td>
<td>B = 1.06*B</td>
</tr>
</tbody>
</table>

**T1: Transfer**
- R(A)
- W(A)

**T2: Interest**
- 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 (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.
Serializable schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>R(B)</td>
<td>R(B)</td>
</tr>
<tr>
<td>W(B)</td>
<td>W(B)</td>
</tr>
<tr>
<td>Commit</td>
<td>Commit</td>
</tr>
</tbody>
</table>
Anomalies with Interleaved Execution

- Not all interleavings of operations are okay.
- Anomaly - two consistency-preserving committed transactions lead to inconsistent state.

- Types of anomalies:
  - Reading Uncommitted Data (WR Conflicts) “dirty reads”
  - Unrepeatable Reads (RW Conflicts)
  - Overwriting Uncommitted Data (WW Conflicts)
### Reading Uncommitted Data (WR Conflict)

<table>
<thead>
<tr>
<th>T1: Transfer</th>
<th>T2: Interest</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

```
R(B)
W(B)
Commit
```

“Dirty Read”
## Unrepeatable Reads (RW Conflicts)

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
<td>Commit</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
<td>Commit</td>
</tr>
</tbody>
</table>
Overwriting Uncommitted Data (WW Conflicts)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>W(B)</td>
<td>W(B)</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>
**Schedules involving abort**

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

- Recoverable schedule: transactions commit only after all transactions whose changes they read commit.
- This is an unrecoverable schedule.
Lock-Based Concurrency Control

- DBMS must ensure
  - only serializable, recoverable schedules are allowed
  - No actions of committed trans lost while undoing aborted trans
- Lock - associated with some object
  - shared or exclusive
- Locking protocol - set of rules to be followed by each transaction to ensure good properties.
Lock Compatibility Matrix

Locks on a data item are granted based on a lock compatibility matrix:

<table>
<thead>
<tr>
<th>Request mode</th>
<th>Mode of Data Item</th>
<th>None</th>
<th>Shared</th>
<th>Exclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Exclusive</td>
<td></td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

When a transaction requests a lock, it must wait (block) until the lock is granted.
Strict Two Phase Locking

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

- **Strict 2PL allows only serializable schedules.**
  - Allows only safe interleavings
  - No anomalies
Schedule following strict 2PL

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>S(A)</td>
<td>S(A)</td>
</tr>
<tr>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>X(B)</td>
<td>W(B)</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
<tr>
<td>X(C)</td>
<td></td>
</tr>
<tr>
<td>R(C)</td>
<td></td>
</tr>
<tr>
<td>W(C)</td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td></td>
</tr>
</tbody>
</table>
Deadlock

- Deadlock must be *prevented* or *avoided*.

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X(B)</td>
<td>X(B)</td>
<td></td>
</tr>
</tbody>
</table>

- granted
- granted
- queued
- queued

- Deadlock must be *prevented* or *avoided*. 

27
Performance of Locking

- Lock-based schemes resolve conflicting schedules by **blocking** and **aborting**
  - in practice few deadlocks and relatively few aborts
  - most of penalty from blocking
- To increase throughput
  - lock smallest objects possible
  - reduce time locks are held
  - reduce hotspots
Transaction support in SQL

- Transaction automatically started for SELECT, UPDATE, CREATE
- Transaction ends with COMMIT or ROLLBACK (abort)
- SQL 99 supports SAVEPOINReTs which are simple nested transactions
What should we lock?

- T1 S-lock on Sailors; T2 X-lock on Sailors
- T1 S-lock on all rows with rating=8; T2 X-lock on Joe’s tuple.
The Phantom Problem

- T1 locks all existing rows with rating=8.
- But a new row satisfying condition could be inserted.
- Phantom problem: A transaction retrieves a collection of tuples and sees different results, even though it did not modify the tuples itself.
  - Conceptually: must lock all possible rows.
  - Can lock entire table.
  - Better, use index locking.
Specify isolation level

- General rules of thumb w.r.t. isolation:
  - Fully serializable isolation is more expensive than “no isolation”
    - We can’t do as many things concurrently (or we have to undo them frequently)
  - For performance, we generally want to specify the most relaxed isolation level that’s acceptable
    - Note that we’re “slightly” violating a correctness constraint to get performance!
Specifying isolation level in SQL

SET TRANSACTION [READ WRITE | READ ONLY]
ISOLATION LEVEL [LEVEL];

LEVEL = SERIALIZABLE
REPEATABLE READ
READ COMMITTED
READ UNCOMMITTED

Less isolation

The default isolation level is SERIALIZABLE

Locks sets of objects, avoids phantoms
REPEATABLE READ

- T reads only changes made by committed transactions
- No value read/written by T is changed by another transaction until T completes.
- Phantoms possible: inserts of qualifying tuples not avoided.

Locks only individual objects
READ COMMITTED

- T reads only changes made by committed transactions
- No value read/written by T is changed by another transaction until T completes.
- Value read by T may be modified while T in progress.
- Phantoms possible.

X locks on written objects, held to end
S locks on read objects, but released immediately.
**READ UNCOMMITTED**

- Greatest exposure to other transactions
- Dirty reads possible
- Can’t make changes: must be READ ONLY
- Does not obtain shared locks before reading
  - Thus no locks every requested.
## Summary of Isolation Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Dirty Read</th>
<th>Unrepeatable Read</th>
<th>Phantoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UN-COMMITTED</td>
<td>Maybe</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</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>
Summary

- Concurrency control and recovery are among the most important functions provided by a DBMS.
- Users need not worry about concurrency.
  - System guarantees nice properties: ACID
  - This is implemented using a locking protocol
- Users can trade isolation for performance using SQL commands
Aborting a Transaction

- If a transaction $Ti$ is aborted, all its actions have to be undone. Not only that, if $Tj$ reads an object last written by $Ti$, $Tj$ must be aborted as well!
- Most systems avoid such cascading aborts by releasing a transaction’s locks only at commit time.
  - If $Ti$ writes an object, $Tj$ can read this only after $Ti$ 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 Xacts at the time of the crash are aborted when the system comes back up.
The Log

- The following actions are recorded in the log:
  - *Ti writes an object*: the old value and the new value.
    - Log record must go to disk *before* the changed page!
  - *Ti commits/aborts*: a log record indicating this action.
- Log records are chained together by Xact id, so it’s easy to undo a specific Xact.
- Log is often *duplexed* and *archived* on stable storage.
- All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.