ICS 321 Fall 2009

Overview of Transaction Management

Asst. Prof. Lipyeow Lim
Information & Computer Science Department
University of Hawaii at Manoa
Transactions

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

• 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 DBMS supports multiple users, ie, multiple transactions may be running concurrently.

• Concurrent executions can be exploited for DBMS performance.
  – Because disk accesses are frequent, and relatively slow, it is important to keep the CPU humming by working on several user programs concurrently.
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 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).

• **Issues:** Effect of *interleaving* transactions, and crashes
ACID Properties

4 important properties of transactions

• **Atomicity**: all or nothing
  – Users regard execution of a transaction as atomic
  – No worries about incomplete transactions

• **Consistency**: a transaction must leave the database in a good state
  – Semantics of consistency is application dependent
  – The user assumes responsibility

• **Isolation**: a transaction is isolated from the effects of other concurrent transaction

• **Durability**: Effects of completed transactions persists even if system crashes before all changes are written out to disk
Atomicity

• A transaction might *commit* after completing all its actions, or it could *abort* (or be aborted by the DBMS) after executing some actions.

• 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.
Example (Atomicity)

- 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.
Example Contd. (Atomicity)

- Consider the following interleavings (schedule)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T1</th>
<th>T2</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = A + 100</td>
<td>A = 1.06 A</td>
<td>A = A + 100</td>
<td>A = 1.06 A</td>
<td>R(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>B = B - 100</td>
<td>B = 1.06 B</td>
<td>B = B - 100</td>
<td>B = 1.06 B</td>
<td>R(A)</td>
<td>W(A)</td>
</tr>
</tbody>
</table>

DBMS’ view of the 2\textsuperscript{nd} schedule
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.

(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)
Anomaly: Dirty Reads

- AKA reading uncommitted Data, WR conflicts

T1          T2
A=A+100     A = 20
A=1.06*A    A = 120
Commit
A = 127.2
B=B-100     A = 20
Abort
With T1 aborted correct value of A = 21.2

T1          T2
R(A)        R(A)
W(A)        W(A)
Commit
R(B)
W(B)
Abort
Anomaly: Phantom Reads

• AKA Unrepeatable Reads, RW conflicts

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Print A</td>
<td>A = 20</td>
</tr>
<tr>
<td>A = 100</td>
<td>A = 20</td>
</tr>
<tr>
<td>Commit</td>
<td>A = 21.2</td>
</tr>
<tr>
<td>Print A</td>
<td>A = 21.2</td>
</tr>
<tr>
<td>A = 1.06*A Commit</td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

T1 sees two different values of A even though T1 did not change A!
Anomaly: Blind Writes

• AKA Overwriting Uncommitted Data, WW conflicts

Can any serializable schedule produce this result?
Lock-based Concurrency Control

- **Strict Two-phase Locking (Strict 2PL) Protocol**:
  - Each Xact 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
    - **(Non-strict) 2PL Variant**: Release locks anytime, but cannot acquire locks after releasing any lock.
  - 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**.
  - Additionally, it simplifies transaction aborts
  - **(Non-strict) 2PL** also allows only serializable schedules, but involves more complex abort processing
**Example (Strict 2PL)**

- Consider the dirty read schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=A+100</td>
<td>A=1.06*A Commit</td>
<td>X(A)</td>
<td>X(A)</td>
</tr>
<tr>
<td>B=B-100</td>
<td>Abort</td>
<td>R(A)</td>
<td>R(A)</td>
</tr>
</tbody>
</table>

**Dirty read on A!**

With Strict 2PL, T2 can only access A when T1 aborts.

A = 20
A = 120
A = 127.2
A = 20
A = 120
A = 127.2

**(11/5/2009)**

Lipyeow Lim -- University of Hawaii at Manoa
Example (Non-Strict 2PL)

- Consider the dirty read schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=A+100</td>
<td>A=20</td>
<td>X(A)</td>
<td>X(A)</td>
</tr>
<tr>
<td>A=1.06*A</td>
<td>A=120</td>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>Commit</td>
<td>Dirty read on A!</td>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>B=B-100</td>
<td>A=127.2</td>
<td>RX(A)</td>
<td>X(B)</td>
</tr>
<tr>
<td>Abort</td>
<td></td>
<td></td>
<td>R(B)</td>
</tr>
</tbody>
</table>

With non-strict 2PL, T2 can still read uncommitted data if T1 aborts!
Deadlocks

• Cycle of transactions waiting for locks to be released
• DBMS has to either prevent or resolve deadlocks
• Common approach:
  – Detect via timeout
  – Resolve by aborting transactions
Aborting a Transaction

• If a transaction $T1$ is aborted, all its actions have to be undone.
  – Not only that, if $T2$ reads an object last written by $T1$, $T2$ must be aborted as well!
• Most systems avoid such cascading aborts by releasing a transaction’s locks only at commit time.
  – If $T1$ writes an object, $T2$ can read this only after $T1$ 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