Introduction to Data Management

Lecture #25
(Transactions: The Final Frontier...!)

Instructor: Mike Carey
mjcarey@ics.uci.edu

Announcements

- HW and exam info:
  - HW#8 now in flight! (Due on Thursday at 5PM)
  - Midterm #2’s solutions are now posted
  - Endterm is in class on Fri, June 8, 5-5:50 PM (!)
    - Cheat sheet allowed, as per usual
    - Non-cumulative (see Wiki syllabus for official scope)
    - Sample exam available (but interpret it appropriately)
    - Will include NoSQL, JSON, and even transactions!

- This week’s material:
  - Today: Physical Design Leftovers, then Transactions
  - Today: Transactions, cont.
First Let's Wrap-Up Physical DB Design!

- Off to a different slide deck for that…

Transactions

- Concurrent execution of user programs is essential for good DBMS performance.
  - Because disk accesses are frequent, and relatively slow, it’s important to keep the CPU cores humming by working on several user programs concurrently.
- A user’s program may carry out many operations on the data pulled from the database, but the DBMS is only concerned about what data is read or written from/to the database.
- A transaction is the DBMS’s abstract view of a user program: a sequence of (record) reads and writes.
The ACID Properties

- **Atomicity**: Each transaction is all or nothing.
  - No worries about partial effects (if failures) and cleanup.
- **Consistency**: Each transaction moves the database from one consistent state to another one.
  - This is largely the application builder’s responsibility.
- **Isolation**: Each transaction can be written as if it’s the only transaction in existence.
  - No concurrency worries while building applications.
- **Durability**: Once a transaction has committed, its effects will not be lost.
  - Application code doesn’t have to worry about data loss.

Concurrency in a DBMS

- Users submit transactions, and they can think of each one as executing all 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 may enforce some ICs, depending on the ICs declared in CREATE TABLE statements. (CHECK, PK/FK, ...)
  - Beyond this, the DBMS does not understand the semantics of the data. (E.g., it doesn’t know how the interest on a bank account is computed.)
- **Issues**: Effect of interleaving transactions, and crashes.
Atomicity of Transactions

- A transaction might *commit* after completing all its actions, or it could *abort* (or be aborted by the DBMS) after executing some actions.
  - Could violate some constraint, encounter some other error, be caught in a crash, or be picked to resolve a deadlock.
- 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

- Consider two transactions (*Xacts*):
  
  ```
  T1: BEGIN A=A+100, B=B-100 END
  T2: BEGIN A=1.06*A, B=1.06*B END
  ```
- Intuitively, the first transaction is transferring $100 from bank account A to bank account B. The second is crediting 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 transactions running serially in some (but either!) order.
A Quick Aside on “A” & “B”

- What are these two transactions, really?

```
T1: BEGIN
    UPDATE Acct SET bal = bal + 100 WHERE acct_no = 101;
    UPDATE Acct SET bal = bal - 100 WHERE acct_no = 201;
END

T2: BEGIN
    UPDATE Acct SET bal = bal * 1.06 WHERE acct_type = 'SV';
END
```

- Again, the first transaction is transferring $100 from account B (201) to account A (101). The second one is giving all savings accounts their 6% interest payment.

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 happens if:

```
T1: A=A+100, B=B-100
T2: A=1.06*A, B=1.06*B ← Too much interest!
```

- The DBMSs 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 (Defn’s.)

- **Serial schedule**: Any 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 (any!) serial execution of the transactions.
  
  *(Note: If each transaction preserves consistency, then every serializable schedule preserves consistency!)*

Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, a.k.a. “dirty reads”):
  
<table>
<thead>
<tr>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A), W(A), R(B), W(B), <strong>Abort</strong></td>
<td></td>
</tr>
<tr>
<td>R(A), W(A), C</td>
<td></td>
</tr>
</tbody>
</table>

- Unrepeatable Reads (RW Conflicts):
  
<table>
<thead>
<tr>
<th>T5</th>
<th>T6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R(A)</strong>, <strong>R(A)</strong>, <strong>W(A)</strong>, <strong>W(A)</strong>, <strong>C</strong></td>
<td></td>
</tr>
<tr>
<td>R(A), W(A), C</td>
<td></td>
</tr>
</tbody>
</table>
Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T7:</strong></td>
<td>W(A), W(B), C</td>
</tr>
<tr>
<td><strong>T8:</strong></td>
<td>W(A), W(B), C</td>
</tr>
</tbody>
</table>

(Note how results are a “must have been concurrent!” intermingling of transactions’ T1 & T2 writes…)

Lock-Based Concurrency Control

- **Strict Two-phase Locking (Strict 2PL) Protocol:**
  - Each Xact must get an *S* (shared) lock on an object before reading, and an *X* (exclusive) lock on it before writing.
  - All locks held by a transaction are released only when the transaction completes.
    - (Non-strict) 2PL Variant: Release locks anytime, but do not acquire any new locks after releasing any lock.
  - **Note:** If a Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object – they must wait.
- **Strict 2PL allows only serializable schedules.**
  - And additionally, it simplifies transaction aborts!
  - (Non-strict) 2PL also allows only serializable schedules, but needs more complex abort processing (as you’ll see).