Introduction to Data Management

Lecture #24
(Transactions)

Instructor: Mike Carey
mjcarey@ics.uci.edu

Announcements

- HW and exam info:
  - HW#8 now in flight! (Due Friday; 5pts/day if late)
  - Endterm Exam: Wed, Mar 22, 11-11:50 AM
- This week’s lecture plan:
  - Today: Transactions
  - Wednesday: **JSON mini-course** (online self-study)
  - Friday: Wrap up + Endterm review! *(Bring Q’s...!)*
- Other notable news:
  - Sample exams are (still 😊) available on wiki page
  - Discussions can include AsterixDB Q&A...
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/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):
  
<table>
<thead>
<tr>
<th>Transaction</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>A = A + 100, B = B - 100</td>
</tr>
<tr>
<td>T2</td>
<td>A = 1.06^*A, B = 1.06^*B</td>
</tr>
</tbody>
</table>

- This is OK. But what happens if:
  
<table>
<thead>
<tr>
<th>Transaction</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>A = A + 100, B = B - 100</td>
</tr>
<tr>
<td>T2</td>
<td>A = 1.06^*A, B = 1.06^*B ← Too much interest!</td>
</tr>
</tbody>
</table>

- The DBMSs view of the second schedule:
  
<table>
<thead>
<tr>
<th>Transaction</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>R(A), W(A), R(B), W(B)</td>
</tr>
<tr>
<td>T2</td>
<td>R(A), W(A), R(B), W(B)</td>
</tr>
</tbody>
</table>

**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”):
  
  | T3:  | R(A), W(A), R(B), W(B), Abort |
  | T4:  | R(A), W(A), C |

- Unrepeatable Reads (RW Conflicts):
  
  | T5:  | R(A), R(A), W(A), C |
  | T6:  | R(A), W(A), C |

Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):
  
  | T7:  | W(A), W(B), C |
  | T8:  | W(A), W(B), C |

(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).

2PL Prevents the Anomalies

- Reading Uncommitted Data (WR Conflicts, a.k.a. “dirty reads”):

  - T3: R(A), W(A), R(B), W(B), Abort
  - T4: R(A), W(A), C

- Unrepeatable Reads (RW Conflicts):

  - T5: R(A), R(A), W(A), C
  - T6: R(A), W(A), C
2PL & Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

  T7: \(W(A), W(B), C\)
  T8: \(W(A), W(B), C\)

  (Now results will no longer be a “must have been concurrent!” intermingling of T1’s & T2’s writes…)

Aborting a Transaction

- If transaction \(Ti\) aborts, all its actions must be undone.
  - And, if some \(Tj\) already read a value last written by \(Ti\), \(Tj\) must also be aborted! (“If I tell you, I’ll have to kill you…” 😊)

- Most systems avoid such cascading aborts by releasing a transaction’s locks only at commit time.
  - If \(Ti\) writes an object, \(Tj\) can read it only after \(Ti\) commits.

- In order to undo the actions of an aborted transaction, the DBMS keeps a log where every write is recorded.
  - Also used to recover from system crashes: active Xacts at crash time are aborted when the DBMS comes back up.
Tune in Friday...!