Transaction Management Overview

Yanlei Diao
UMass Amherst
March 15, 2007

DBMS Architecture

- Query Parser
- Query Rewriter
- Query Optimizer
- Query Executor
- Lock Manager
- Access Methods
- Buffer Manager
- Log Manager
- Disk Space Manager
- DB

- Declarative, Expressive language
- Efficient Execution
- Concurrency Control
- Recovery

Concurrent User Programs

- Concurrent execution of user programs is essential for good DBMS performance.
  - Disk accesses (I/O) are frequent and relatively slow, so it is important to keep the CPU humming by working on several user programs concurrently.
  - Good for average response time.
  - Good for system throughput.
Transactions

- A user’s program may do many things on the data retrieved.
  - E.g., operations on Bob’s bank account.
- But the DBMS is only concerned about what data is read from/written to the database.
- A transaction is the DBMS’s abstract view of a user program: a sequence of reads and writes.
  - E.g., transfer of money from account A to account B, including debiting A and crediting B.

ACID Properties of Transactions

- Many users submit transactions, but each can think of his transaction as executing by itself.
  - Concurrency is achieved by the DBMS, which interleaves reads and writes of various transactions.
- Consistency: each transaction starts and ends with a consistent state; i.e. it can’t break ICs.
  - E.g., if an IC states that all accounts must have a positive balance, no transaction can violate this rule.
- Isolation: execution of one transaction appears isolated from others.
  - Nobody else can see the data in its intermediate state, e.g. account A being debited but B not being credited.

ACID Properties (Contd.)

- A transaction might commit after completing all its actions, or it could abort (or be aborted by the DBMS) after executing some actions.
- Atomicity: either all actions of a transaction are performed or none of them is (all-or-none).
  - DBMS logs all actions so that it can undo the actions of aborted transactions.
- Durability: once a user program has been notified of success, its effect will persist despite system failure.
  - DBMS logs all actions so that it can redo the actions of committed 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 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.)

- 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 about:
  
  T1: A=A+100, B=B-100  
  T2: A=1.06*A, B=1.06*B 

- The DBMS’s 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

- 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 DB) 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.)
Anomalies with Interleaved Execution

- **Reading Uncommitted Data** (WR Conflicts, “dirty reads”):
  
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A), W(A), R(B), W(B), Abort</td>
<td>R(A), W(A), C</td>
</tr>
</tbody>
</table>

- **Unrepeatable Reads** (RW Conflicts):
  
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A),</td>
<td>R(A), W(A), C</td>
</tr>
<tr>
<td></td>
<td>R(A), W(A), C</td>
</tr>
</tbody>
</table>

Anomalies (Contd.)

- **Overwriting Uncommitted Data** (WW Conflicts):
  
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>W(A), W(B), C</td>
<td>W(A), W(B), C</td>
</tr>
</tbody>
</table>

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.
  - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
  - 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.
Properties of 2PL

- **Strict 2PL:**
  - Allows only serializable schedules.
  - Additionally, simplifies transaction aborts.

- **(Non-strict) 2PL:**
  - Also allows only serializable schedules.
  - But involves more complex abort processing.

Aborting a Transaction

- **Abort:** If a transaction $T_i$ is aborted, all its actions have to be undone.
- **Cascading aborts:** Not only that, if $T_j$ reads an object last written by $T_i$, $T_j$ must be aborted as well!
- Most systems avoid cascading aborts by releasing a transaction’s locks only at commit time (strict 2PL).
  - If $T_i$ writes an object, $T_j$ can read this only after $T_i$ commits.
- To *undo* the actions of an aborted transaction, DBMS maintains a log in which every write is recorded.
  - Also used to recover from system crashes: all active Xacts at the time of crash are aborted when system comes back up.

The Log

- The following actions are recorded in the log:
  - $T_i$ writes an object: the old value and the new value.
  - Log record must go to disk before the changed page!
  - $T_i$ 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 are handled transparently by the DBMS.
Recovering From a Crash

- Three phases in the Aries recovery algorithm:
  - **Analysis**: Scan the log forward (from the most recent checkpoint) to identify all Xacts that were active, and all dirty pages in the buffer pool at the time of the crash.
  - **Redo**: Redoes all updates to dirty pages in the buffer pool, as needed, to ensure that all logged updates are in fact carried out and written to disk.
  - **Undo**: The writes of all Xacts that were active at the crash are undone (by restoring the before value of the update, which is in the log record for the update), working backwards in the log. (Care must be taken to handle the case of a crash occurring during the recovery process!)

Summary

- Concurrency control and recovery are among the most important functions provided by a DBMS.
- Users need not worry about concurrency:
  - System automatically inserts lock/unlock requests and schedules 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.
- Write-ahead logging (WAL) is used to undo the actions of aborted transactions and to restore the system to a consistent state after a crash.
  - **Consistent state**: Only the effects of committed Xacts seen.