Transaction Management Overview

There are three side effects of acid. Enhanced long term memory, decreased short term memory, and I forget the third.

- Timothy Leary

Concurrency Control & Recovery

- Concurrency Control
  - Provide correct and highly available data access in the presence of concurrent access by many users
- Recovery
  - Ensures database is fault tolerant, and not corrupted by software, system or media failure
  - 24x7 access to mission critical data
- A boon to application authors!
  - Existence of CC&R allows applications to be written without explicit concern for concurrency and fault tolerance

Roadmap

- Overview (Today)
- Concurrency Control (1-2 lectures)
- Recovery (1-2 lectures)

Structure of a DBMS

These layers must consider concurrency control and recovery (Transaction, Lock, Recovery Managers)

Transactions and Concurrent Execution

- Transaction ("xact") - DBMS's abstract view of a user program (or activity):
  - A sequence of reads and writes of database objects.
  - Unit of work that must commit or abort as an atomic unit
- Transaction Manager controls the execution of transactions.
- User’s program logic is invisible to DBMS!
  - Arbitrary computation possible on data fetched from the DB
  - The DBMS only sees data read/written from/to the DB.
- Challenge: provide atomic transactions to concurrent users!
  - Given only the read/write interface.

Concurrency: Why bother?

- The latency argument
- The throughput argument
- Both are critical!
ACID properties of Transaction Executions

• **Atomicity**: All actions in the Xact happen, or none happen.
• **Consistency**: If each Xact is consistent, and the DB starts consistent, it ends up consistent.
• **Isolation**: Execution of one Xact is isolated from that of other Xacts.
• **Durability**: If a Xact commits, its effects persist.

Atomicity and Durability

• A transaction ends in one of two ways:
  – *commit* after completing all its actions
  – *abort* (or be aborted by the DBMS) after executing some actions,
    – Or *system crash* while the xact is in progress; treat as abort.
• Two important properties for a transaction:
  – Atomicity: Either execute all its actions, or none of them
  – Durability: The effects of a committed xact must survive failures.
• DBMS ensures the above by *logging* all actions:
  – *Undo* the actions of aborted/failed transactions.
  – *Redo* actions of committed transactions not yet propagated to disk when system crashes.

Transaction Consistency

• Transactions preserve DB consistency
  – Given a consistent DB state, produce another consistent DB state
• DB Consistency expressed as a set of declarative Integrity Constraints
  – CREATE TABLE/ASSERTION statements
  – E.g. Each CS186 student can only register in one project group. Each group must have 2 students.
  – Application-level
  – E.g. Bank account total of each customer must stay the same during a "transfer" from savings to checking account
• Transactions that violate ICs are aborted
  – That's all the DBMS can automatically check!

Isolation (Concurrency)

• DBMS interleaves actions of many xacts concurrently
  – Actions = reads/writes of DB objects
• DBMS ensures xacts do not "step onto" one another.
• Each xact executes as if it were running by itself.
  – Concurrent accesses have no effect on a Transaction's behavior
  – Net effect *must be* identical to executing all transactions for some serial order.
  – Users & programmers think about transactions in isolation
    – Without considering effects of other concurrent transactions!

Example

• Consider two transactions *(Xacts)*:

<table>
<thead>
<tr>
<th>Xact</th>
<th>Action 1</th>
<th>Action 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>A=A+100</td>
<td>B=B-100</td>
</tr>
<tr>
<td>T2</td>
<td>A=1.06*A</td>
<td>B=1.06*B</td>
</tr>
</tbody>
</table>

• 1st xact transfers $100 from B's account to A's
• 2nd credits both accounts with 6% interest.
• Assume at first A and B each have $1000. What are the legal outcomes of running T1 and T2?
  - T1 ; T2 *(A=1166, B=954)*
  - T2 ; T1 *(A=1160, B=960)*
• In either case, A+B = $2000 *1.06 = $2120
• There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together.

Example (Contd.)

• Consider a possible interleaved *schedule*.

<table>
<thead>
<tr>
<th>Xact</th>
<th>Action 1</th>
<th>Action 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>A=A+100, B=100</td>
<td>B=100</td>
</tr>
<tr>
<td>T2</td>
<td>A=1.06<em>A, B=1.06</em>B</td>
<td></td>
</tr>
</tbody>
</table>

• This is OK (same as T1;T2). But what about:

<table>
<thead>
<tr>
<th>Xact</th>
<th>Action 1</th>
<th>Action 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>A=A+100, B=100</td>
<td>B=100</td>
</tr>
<tr>
<td>T2</td>
<td>A=1.06<em>A, B=1.06</em>B</td>
<td></td>
</tr>
</tbody>
</table>

• Result: A=1166, B=960; A+B = 2126, bank loses $6!
• The DBMS's view of the second schedule:

<table>
<thead>
<tr>
<th>Xact</th>
<th>Action 1</th>
<th>Action 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>R(A), W(A)</td>
<td>R(B), W(B)</td>
</tr>
<tr>
<td>T2</td>
<td>R(A), W(A), R(B), W(B)</td>
<td></td>
</tr>
</tbody>
</table>
**Scheduling Transactions: Definitions**

- **Serial schedule:** no concurrency
  - Does not interleave the actions of different transactions.
- **Equivalent schedules:** same result on any DB state
  - 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:** equivalent to a serial 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”):**
  - T1: R(A), W(A), R(B), W(B), Abort
  - T2: R(A), W(A), C

- **Unrepeatable Reads (RW Conflicts):**
  - T1: R(A), R(B), W(A), C
  - T2: R(A), W(A), C

**Anomalies (Continued)**

- **Overwriting Uncommitted Data (WW Conflicts):**
  - T1: W(A), W(B), C
  - T2: W(A), W(B), C

**Lock-Based Concurrency Control**

- **A simple mechanism to allow concurrency but avoid the anomalies just described...**
- **Two-phase Locking (2PL) Protocol:**
  - Always obtain a S (shared) lock on object before reading.
  - Always obtain 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.
  - DBMS internally enforces the above locking protocol.
  - Two phases: acquiring locks, and releasing them.
    - No lock is ever acquired after one has been released.
    - “Growing phase” followed by “shrinking phase.”
- **Lock Manager** tracks lock requests, grants locks on database objects when they become available.

**Strict 2PL**

- 2PL allows only serializable schedules but is subjected to cascading aborts.
- Example: rollback of T1 requires rollback of T2!
  - T1: R(A), W(A), Abort
  - T2: R(A), W(A), R(B), W(B)
- To avoid Cascading aborts, use Strict 2PL
- **Strict Two-phase Locking (Strict 2PL) Protocol:**
  - Same as 2PL, except:
    - A transaction releases no locks until it completes.

**Introduction to Crash Recovery**

- **Recovery Manager**
  - Upon recovery from crash:
    - Must bring DB to a consistent transactional state.
  - Ensures transaction Atomicity and Durability.
  - Undoes actions of transactions that do not commit.
  - Redoes lost actions of committed transactions.
    - lost during system failures or media failures.
- **Recovery Manager** maintains log information during normal execution of transactions for use during crash recovery.
The Log

- Log consists of "records" that are written sequentially.
  - Stored on a separate disk from the DB
  - Typically chained together by Xact id
  - Log is often *duplexed and archived* on stable storage.

- Log stores modifications to the database
  - *If Ti writes an object, write a log record with:*
    - If UNDO required need "before image"
    - If REDO required need "after image"
  - *Ti commits/aborts: a log record indicating this action.*

- Need for UNDO/REDO depend on Buffer Mgr (!!)
  - UNDO required if uncommitted data can overwrite stable version of committed data (*STEAL* buffer management).
  - REDO required if xact can commit before all its updates are on disk (*NO FORCE* buffer management).

Logging Continued

- Write Ahead Logging (WAL) protocol
  - Log record must go to disk before the changed page!
  - Implemented via a handshake between log manager and the buffer manager.
  - All log records for a transaction (including its commit record) must be written to disk before the transaction is considered "Committed".

- All log related activities are handled transparently by the DBMS.
  - As was true of CC-related activities such as lock/unlock, dealing with deadlocks, etc.

ARIES Recovery

- There are 3 phases in ARIES recovery protocol:
  - *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, as found in the log), working backwards in the log.

- At the end --- all committed updates and only those updates are reflected in the database.

- Some 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.

- Concurrency control (Isolation) is automatic.
  - DBMS issues proper Two-Phase Locking (ZPL) requests
  - Enforces lock discipline (S & X)
  - End result promised to be "serializable": equivalent to some serial schedule

- Atomicity and Durability ensured by Write-Ahead Logging (WAL) and recovery protocol
  - Used to undo the actions of aborted transactions (no subatomic stuff visible after recovery!)
  - Used to redo the lost actions of committed transactions