Transaction Processing

Overview

Concept of a Transaction

- ATM machine: balance transfer
  - Transfer $100 from Savings to Checking
- What goes on in the Database
  - SQL query to fetch current balance from savings
  - SQL query to fetch current balance from checking
  - Update balance in Savings
  - Update balance in checking
- What is your view of what goes on?
  - One step or many steps?
Transactions

- A *transaction* is the DBMS’s abstract view of a user program
- *Transaction* (Xact).
  - A sequence of many actions considered to be one atomic unit of work submitted to the DBMS.
  - Execute all actions in Xact or none of them
  - DBMS point of view: sequence of reads and writes
- DBMS “actions”:
  - reads, writes
  - Special actions: commit, abort
  - for now, assume reads and writes are on tuples;
  - Model Xact operations only as Read R(A) and Write W(A)

Concurrency: Why?

- What if multiple users want to access the same database?
  - Banner, bank, airline reservation system, ....
- Performance: Better transaction throughput, response time
  - While one processes is doing a disk read, another can be using the CPU or reading another disk.
- But…..
- DANGER DANGER! Concurrency could lead to incorrectness!
  - Must carefully manage concurrent data access.
  - There’s (much!) more here than the usual OS tricks!
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.

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.

- A very important property guaranteed by the DBMS for all transactions is that they are atomic.
  - 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.
The Committed Xact

- Xact enters committed state if it has partially committed (i.e., last statement executed) and it is guaranteed that it is never aborted
  - This must be ensured by the recovery system!
  - This is the "semantics" of the commit instruction

Failures in DBMS

- Computer system subject to failure
  - Power failure: lose contents of main memory
  - Disk crash: could lose non-volatile storage
  - Software errors
- When failure occurs - info related to DB is lost
- Recovery scheme responsible for handling failures and restoring database to consistent state
The ACID Properties

- **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.

Passing the ACID Test

- **Concurrency Control**
  - Guarantees Consistency and Isolation, given Atomicity.
- **Logging and Recovery**
  - Guarantees Atomicity and Durability.
  - Log file based recovery techniques
Transaction States

- Aborted Xact should have no effect on DB
  - Therefore database must be rolled back to restore to the state before Xact
- Xact that successfully completes must be committed and database updated accordingly
  - Instruction to specify “commit”
- Effect of committed Xact cannot be undone by aborting Xact, must use a compensating Xact

Transaction States

- Active: initial state
- Partially Committed: after last statement
- Failed: after discovering normal exec cannot proceed
- Aborted: after Xact rolled back and DB restored to original state
- Committed: after successful completion
  - Can represent above by a finite state diagram
State Diagram

Log-based Recovery Systems

- Concept: keep a log (i.e., record) of the states of the Xact and refer to log to determine how to update database to maintain consistency
- Normal operation: reflect all changes \textit{first} in log file
- Recovering from failure: look up log file to figure out next step
**Concept: Log-based Recovery**

- Always write to a log file first before writing to database
  - Enter log records
- Transaction states:
  - Commit, Abort, Start
- Redo and Undo Operations
  - Redo Xact (write new values) if commit record in log
  - Undo Xact (restore old values) if no commit
- Deferred Modification or Immediate
  - Allow DB Write only after commit record written vs. allow write immediately after record written into log file

**Concurrency: Schedules**

- Scheduler is a program (in the Operating System) that controls concurrent execution of Xacts; it produces execution sequence for a set of Xacts
  - Schedule
- Schedule must preserve instruction execution order
- **Serial schedule** is when transactions are executed sequentially
Example …

- Consider two transactions (Xacts):
  
  \[
  \begin{align*}
  \text{T1:} & \quad \text{BEGIN } A=A+100, \ B=B-100 \ \text{END} \\
  \text{T2:} & \quad \text{BEGIN } A=1.06 \times A, \ B=1.06 \times B \ \text{END}
  \end{align*}
  \]

- 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.

Serial execution:

- Assume initial A=1000, B=1000
- T1 followed by T2
  - After T1: A=900, B=1100
  - After T2: A= 1.06\times900= 954, B= 1.06\times1100 = 1166
  - Total: T1+T2 = 2120
- T2 followed by T1
  - After T2: A= 1.06\times1000 = 1060, B= 1.06\times1000=1060
  - After T1: A= 960 B=1160
  - Total: T1+T2 = 2120
Example (Contd.)

- Consider a possible interleaving (schedule):

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=A+100,</td>
<td>A=1.06*A,</td>
</tr>
<tr>
<td>B=B-100</td>
<td>B=1.06*B</td>
</tr>
</tbody>
</table>

Is this schedule ok?

Example (Contd.)

- Previous schedule OK. But what about:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=A+100,</td>
<td>A=1.06*A,</td>
</tr>
<tr>
<td>B=B-100</td>
<td>B=1.06*B</td>
</tr>
</tbody>
</table>

- The DBMS’s view of the second schedule:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A), W(A),</td>
<td>R(A), W(A), R(B), W(B)</td>
</tr>
<tr>
<td>R(B), W(B)</td>
<td>R(A), W(A), R(B), W(B)</td>
</tr>
</tbody>
</table>
Schedules

- **Schedule**: An interleaving of actions from a set of Xacts, where the actions of any 1 Xact are in the original order.
  - Represents some actual sequence of database actions.
  - Example: \( R_1(A), W_1(A), R_2(B), W_2(B), R_1(C), W_1(C) \)
  - In a **complete** schedule, each Xact ends in commit or abort.

- Initial State + Schedule → Final State

Acceptable Schedules

- One sensible “isolated, consistent” schedule:
  - Run Xacts one at a time, in a series.
  - This is called a **serial** schedule.
  - **NOTE**: Different serial schedules can have different final states; all are “OK” -- DBMS makes no guarantees about the order in which concurrently submitted Xacts are executed.

- Serial schedule always produces correct results
  - How can we use this to construct a concurrency theory to prove correctness?

- Note: Initial State + Schedule → Final State
Serializable Schedules

- **Serializable** schedules:
  - Final state is what some serial schedule would have produced.
  - Aborted Xacts are not part of schedule; ignore them for now (they are made to “disappear” by using logging).
- Serializable schedule produces correct results

Generating correct schedules

- Generate schedule that gives the same result as some serial schedule
- How?
- Provide system level primitives that force only serializable schedules
Serializability Violations

- Two actions **conflict** when 2 Xacts access the same item:
  - W-R conflict: T2 reads something T1 wrote.
  - R-W and W-W conflicts: Similar.
- WR conflict (dirty read):
  - Result is not equal to any serial execution!
- RR Conflict – is this a problem?

<table>
<thead>
<tr>
<th></th>
<th>R(A)</th>
<th>W(A)</th>
<th>R(A)</th>
<th>W(A)</th>
<th>R(B)</th>
<th>W(B)</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>R(A)</td>
<td>W(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td></td>
<td></td>
<td>R(B)</td>
<td>W(B)</td>
<td></td>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

Now, Aborted Transactions

- **Serializable schedule**: Equivalent to a serial schedule of committed Xacts.
  - as if aborted Xacts never happened.
- Two Issues:
  - How does one undo the effects of an xact?
    - Covered by logging/recovery process
  - What if another Xact sees these effects??
    - Must undo that Xact as well!
Cascading Aborts

- Abort of T1 requires abort of T2!
  - Cascading Abort
- What about WW conflicts & aborts?
  - T2 overwrites a value that T1 writes.
  - T1 aborts: its “remembered” values are restored.
  - Lose T2’s write! We will see how to solve this, too.
- An ACA (avoids cascading abort) schedule is one in which cascading abort cannot arise.
  - A Xact only reads/writes data from committed Xacts.

Recoverable Schedules

- Abort of T1 requires abort of T2!
  - But T2 has already committed!
- A recoverable schedule is one in which this cannot happen.
  - i.e. a Xact commits only after all the Xacts it “depends on” (i.e. it reads from or overwrites) commit.
  - Recoverable implies ACA (but not vice-versa!).
  - Find example ...
- Real systems typically ensure that only recoverable schedules arise
Recap: Scheduling Concurrent Transactions

- Need to ensure ACID properties
- Schedule must produce results equivalent to a serial schedule (avoid conflicting R/W)
  - Guarantee serializability
- Must be able to recover from a failure
  - Guarantee recoverability and avoid cascading aborts
- What type of schedules??
  - Recoverable serializable schedules

Mechanism for generating correct schedules

- System must only allow recoverable serializable schedules
- How?
  - Problem: when different Xacts access same data
  - Solution: control access to shared data
    - When T1 is changing data, nobody else can access this data
      - T1 needs to place a LOCK on the data
Correct Schedules?

- Interleaving of instructions can lead to incorrect results
- What is a correct schedule – reasoning about correctness
- Serializability theory – easier to reason about serial executions than concurrent schedules
  - If results of a schedule are identical to results of a serial schedule then?
- Note: we need only focus on Read and Write operations to common data when modeling concurrency

Concept of Locks: Concurrency Control

- Goal: Generate “correct” schedules
Mechanism for generating correct schedules

- System must only allow recoverable serializable schedules
- How?
  - Problem: when different Xacts access same data
  - Solution: control access to shared data
    - When T1 is changing data, nobody else can access this data
    - T1 needs to place a LOCK on the data

Lock Based Protocols

- Conflict occurs when two Xacts try to access the same data item
- Associate a “lock” for each shared data item
  - Similar to mutual exclusion
  - To access a data item, check if it is unlocked else wait
  - Need to worry about the type of operation: Read or Write
    - Leads to Lock Modes: Shared Lock(S) for Reads only and Exclusive Lock(X) for Writes
**Locks**

T1: start
lock(A)
Read(A)
Unlock(A)
lock(B)
Write(B)
Unlock(B)
commit

---

**Lock Based Protocols**

- Conflict occurs when two Xacts try to access the same data item
- Associate a “lock” for each shared data item
  - Similar to mutual exclusion
  - To access a data item, check if it is unlocked else wait
  - Need to worry about the type of operation: Read or Write
    - Leads to Lock Modes: Shared Lock(S) for Reads only and Exclusive Lock(X) for Writes
Strict 2PL

- **Strict 2PL:**
  - If T wants to read an object, first obtains an S lock.
  - If T wants to modify an object, first obtains X lock.
  - Hold all locks until end of transaction.
- Guarantees serializability, and recoverable schedule, too!

![Graph showing # of locks over time]

More on Concurrency and recovery

- Operating Systems