Relational Transaction Processing

CS 2541: Database Systems

Course Summary

- Relational Model
  - Design, query languages, app development
- File organization – how data is physically stored
  - Indexing, sorting,…
  - Applies to any storage/DB model
- Implementing some relational operators
  - Algorithms using different file organizations
- Finally: “real world” – users interact with DBMS by submitting transactions
  - Banking, Course registration, Purchases,…

Advanced Topics

- What is transaction processing
  - Concept of transactions, ACID properties
- Data analytics – analyzing data
  - Data mining
- Motivation for NoSQL databases

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?
  - 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.
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
- Key concept behind concurrency theory:
  - Serial schedule leads to correct results, therefore any schedule whose results are equivalent to a serial schedule is a correct schedule
  - Serializability of schedules

Example ...

- Consider two transactions (Xacts):
  
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN A=A+100, B=B-100 END</td>
<td>BEGIN A=1.10<em>A, B=1.10</em>B END</td>
</tr>
</tbody>
</table>

  - Intuitively, the first transaction is transferring $100 from B’s account to A’s account. The second is crediting both accounts with a 10% 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 Schedule and “correct” results

- Serial execution: Assume initial A=1000, B=1000
- T1 followed by T2
  - T1 transfers $100 between accts; T2 adds 10% interest
  - After T1: A=900, B=1100
  - After T2: A=1.10*900=990, B=1.10*1100=1210
  - Total: T1+T2 = 2200
- T2 followed by T1
  - After T2: A=1.10*1000=1100, B=1.10*1000=1100
  - After T1: A=1000 B=1200
  - Total: T1+T2 = 2200

  - Therefore any interleaving of operations that result in same T1+T2 as above is a “correct schedule”

Concurrent (Interleaved schedules)

- Operations from different transactions/programs are interleaved to get a concurrent schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=A+100, B=B-100</td>
<td>A=1.10<em>A, B=1.10</em>B</td>
</tr>
</tbody>
</table>
Correct Concurrent schedule

- Through time: starting values \( A=1000, B=1000 \)
  - \( T_1 \) does \( A = A + 100; A = 1100 \)
  - \( T_2 \) does \( A = 1.10^*A = 1.10^* 1100 = 1210 \)
  - \( T_1 \) does \( B=B-100; B= 900 \)
  - \( T_2 \) does \( B = 1.10^*B = 1.10^* 900 = 990 \)
  - Final \( T_1 + T_2 = 1210 + 900 = 2200 \)
  - Therefore correct schedule

| \( T_1 \): | \( A=A+100, \) | \( B=B-100 \) |
| \( T_2 \): | \( A=1.10^*A, \) | \( B=1.10^*B \) |

Incorrect Concurrent schedule

- Through time: starting values \( A=1000, B=1000 \)
  - \( T_1 \) does \( A = A + 100; A = 1100 \)
  - \( T_2 \) does \( A = 1.10^*A = 1.10^* 1100 = 1210 \)
  - \( T_2 \) does \( B= 1.10^*B; B = 1100 \)
  - \( T_1 \) does \( B = B - 100; 1100 - 100 = 1000 \)
  - Final \( T_1 + T_2 = 1210 + 1000 = 2210 \)
  - Therefore incorrect schedule

| \( T_1 \): | \( A=A+100, \) | \( B=B-100 \) |
| \( T_2 \): | \( A=1.10^*A, \) | \( B=1.10^*B \) |

Example of incorrect schedule – what went wrong?

- \( T_1 \): \( A = A + 100, \) \( B = B - 100 \)
- \( T_2 \): \( A = 1.10^*A, B = 1.10^*B \)

- The DBMS’s view of the schedule:

| \( T_1 \): | \( R(A), W(A), \) | \( R(B), W(B) \) |
| \( T_2 \): | \( R(A), W(A), R(B), W(B) \) |

T2 read data changed by T1 before T1 wrote new value of B if T1 fails, then even bigger problem

Problem with incorrect schedule

- T2 read data changed by T1, but before T1 had completed changing the value of B
- Therefore: we should prevent T2 from reading/changing data that T1 is still working on
- Need to isolate T1 and T2
- How?
- Concept of LOCKS
  - Lock the data you need to work on
  - Once done, release the lock and let others read 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

Concurrency and Recoverability: In Operating Systems

- You will study concepts to support concurrency
- Locks, Semaphores, etc.

- Log based recovery to recover from system failures
  - Write your data to a log file
  - If system crashes, read from log file to rollback to a consistent state

Locks

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

Next: Moving away from Relational Model