Transactions; Concurrency; Recovery

Amol Deshpande
CMSC424
Spring 2020 – Online Instruction Plan

- Week 1: File Organization and Indexes
- Week 2: Query Processing
- Week 3: Query Optimization; Parallel Databases 1
- Week 4: Parallel Databases; Mapreduce; Transactions 1
  - Map-reduce and Apache Spark
  - Parallel Databases 2: Execution and Other Issues
  - Transactions 1: ACID, SQL Transactions
  - Homework Due April 24
- Week 5: Transactions 2 (Homework Due May 1)
- Week 6: Miscellaneous Topics (Reading Homework Due May 8)
Transactions: Overview

Book Chapters


Key topics:

- Transactions and ACID Properties
- Different states a transaction goes through
- Notion of a ”Schedule”
- Introduction to Serializability
Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.

- E.g. transaction to transfer $50 from account A to account B:
  1. `read(A)`
  2. `A := A – 50`
  3. `write(A)`
  4. `read(B)`
  5. `B := B + 50`
  6. `write(B)`

- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions
### Overview

- **Transaction**: A sequence of database actions enclosed within special tags

- **Properties:**
  - **Atomicity**: Entire transaction or nothing
  - **Consistency**: Transaction, executed completely, takes database from one consistent state to another
  - **Isolation**: Concurrent transactions *appear* to run in isolation
  - **Durability**: Effects of committed transactions are not lost

- Consistency: Transaction programmer needs to guarantee that
  - DBMS can do a few things, e.g., enforce constraints on the data

- Rest: DBMS guarantees
How does.. this relate to *queries* that we discussed?

⭐ Queries don’t update data, so *durability* and *consistency* not relevant

⭐ Would want *concurrency*
  ➢ Consider a query computing total balance at the end of the day

⭐ Would want *isolation*
  ➢ What if somebody makes a *transfer* while we are computing the balance
  ➢ Typically not guaranteed for such long-running queries

TPC-C vs TPC-H
Assumptions and Goals

Assumptions:

- The system can crash at any time
- Similarly, the power can go out at any point
  - Contents of the main memory won’t survive a crash, or power outage
- BUT… disks are durable. They might stop, but data is not lost.
  - For now.
- Disks only guarantee atomic sector writes, nothing more
- Transactions are by themselves consistent

Goals:

- Guaranteed durability, atomicity
- As much concurrency as possible, while not compromising isolation and/or consistency
  - Two transactions updating the same account balance… NO
  - Two transactions updating different account balances… YES
Transaction states

- **Initial State** – stays in this during execution
- **Successfully completed**
- **Failed**
- **Aborted**

Any changes have been rolled back.
Concurrency: Why?
- Increased processor and disk utilization
- Reduced average response times

Concurrency control schemes
- A CC scheme is used to guarantee that concurrency does not lead to problems
- For now, we will assume durability is not a problem
  - So no crashes
  - Though transactions may still abort

Schedules

When is concurrency okay?
- Serial schedules
- Serializability
A Schedule

Transactions:
 T1: transfers $50 from A to B
 T2: transfers 10% of A to B
Database constraint: A + B is constant (*checking*+*saving accts*)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>Effect:</td>
</tr>
<tr>
<td>A = A -50</td>
<td>Before</td>
</tr>
<tr>
<td>write(A)</td>
<td>A</td>
</tr>
<tr>
<td>read(B)</td>
<td>B</td>
</tr>
<tr>
<td>B = B + 50</td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
</tr>
</tbody>
</table>

Each transaction obeys the constraint.

This schedule does too.
A schedule is simply a (possibly interleaved) execution sequence of transaction instructions.

**Serial Schedule:** A schedule in which transaction appear one after the other

- i.e., No interleaving

Serial schedules satisfy isolation and consistency

- Since each transaction by itself does not introduce inconsistency
Example Schedule

- Another “serial” schedule:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>Effect:</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>tmp = A*0.1</td>
<td>A</td>
<td>100</td>
<td>40</td>
</tr>
<tr>
<td>A = A - tmp</td>
<td>write(A)</td>
<td>B</td>
<td>50</td>
<td>110</td>
</tr>
<tr>
<td>read(B)</td>
<td>tmp = A*0.1</td>
<td>A</td>
<td>100</td>
<td>40</td>
</tr>
<tr>
<td>A = A - tmp</td>
<td>write(A)</td>
<td>B</td>
<td>50</td>
<td>110</td>
</tr>
<tr>
<td>read(B)</td>
<td>B = B+ tmp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B = B+ tmp</td>
<td>write(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(A)</td>
<td>A = A -50</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B=B+50</td>
<td>write(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Consistent?

Constraint is satisfied.

Since each Xion is consistent, any serial schedule must be consistent.
Another schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>A = A - 50</td>
<td>tmp = A*0.1</td>
</tr>
<tr>
<td>write(A)</td>
<td>A = A – tmp</td>
</tr>
<tr>
<td>read(B)</td>
<td>write(A)</td>
</tr>
<tr>
<td>B = B + 50</td>
<td>write(B)</td>
</tr>
</tbody>
</table>

Is this schedule okay?

Let's look at the final effect...

Effect:  

<table>
<thead>
<tr>
<th></th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100</td>
<td>45</td>
</tr>
<tr>
<td>B</td>
<td>50</td>
<td>105</td>
</tr>
</tbody>
</table>

Consistent.  
So this schedule is okay too.
Another schedule

T1
read(A)
A = A - 50
write(A)

read(B)
B = B + 50
write(B)

T2
read(A)
tmp = A * 0.1
A = A - tmp
write(A)

read(B)
B = B + tmp
write(B)

Is this schedule okay?

Let's look at the final effect...

Effect:

<table>
<thead>
<tr>
<th></th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100</td>
<td>45</td>
</tr>
<tr>
<td>B</td>
<td>50</td>
<td>105</td>
</tr>
</tbody>
</table>

Further, the effect same as the serial schedule 1.

Called *serializable*
A “bad” schedule

T1
read(A)
A = A -50
write(A)
read(B)
write(B)

T2
read(A)
tmp = A*0.1
A = A – tmp
write(A)
read(B)

Effect:

<table>
<thead>
<tr>
<th></th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>B</td>
<td>50</td>
<td>60</td>
</tr>
</tbody>
</table>

Not consistent
Serializability

- A schedule is called *serializable* if its final effect is the same as that of a *serial schedule*

- Serializability $\rightarrow$ schedule is fine and doesn’t cause inconsistencies
  - Since serial schedules are fine

- Non-serializable schedules unlikely to result in consistent databases

- We will ensure serializability
  - Typically relaxed in real high-throughput environments

- Not possible to look at all $n!$ serial schedules to check if the effect is the same
  - Instead we ensure serializability by allowing or not allowing certain schedules
### Example Schedule with More Transactions

<table>
<thead>
<tr>
<th></th>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
<th>T₄</th>
<th>T₅</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>read(Y)</td>
<td>read(X)</td>
<td>read(Y)</td>
<td>read(V)</td>
<td>read(V)</td>
</tr>
<tr>
<td></td>
<td>read(Z)</td>
<td>read(Y)</td>
<td>write(Y)</td>
<td>read(V)</td>
<td>read(W)</td>
</tr>
<tr>
<td>T₁</td>
<td>read(U)</td>
<td>write(Y)</td>
<td>write(Z)</td>
<td>read(Y)</td>
<td>read(Z)</td>
</tr>
<tr>
<td></td>
<td>read(U)</td>
<td>write(U)</td>
<td></td>
<td>write(Y)</td>
<td>read(Z)</td>
</tr>
<tr>
<td>T₂</td>
<td>read(Y)</td>
<td></td>
<td></td>
<td>write(Y)</td>
<td></td>
</tr>
<tr>
<td>T₂</td>
<td>read(Z)</td>
<td></td>
<td></td>
<td></td>
<td>read(Z)</td>
</tr>
<tr>
<td>T₃</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>read(Z)</td>
</tr>
</tbody>
</table>
Transactions is how we update data in databases

ACID properties: foundations on which high-performance transaction processing systems are built
★ From the beginning, consistency has been a key requirement
★ Although “relaxed” consistency is acceptable in many cases (originally laid out in 1975)

NoSQL systems originally eschewed ACID properties
★ MongoDB was famously bad at guaranteeing any of the properties
★ Lot of focus on what’s called “eventual consistency”

Recognition today that strict ACID is more important than that
★ Hard to build any business logic if you have no idea if your transactions are consistent