Lecture 8--Continued: Concurrency & Locking
Announcements

• Scores were quite good overall for homework! We’re excited!
  • Destroy the midterm!

• Midterm is with CAs.
  • We will post on the page how to divide into overflow rooms
  • Please start posting questions (some very good ones already!)
  • I promise to be there for final CA

• Trolling: no SQL and bitcoin (OPTIONAL!) bitcoin exchange brought down by lack of consistency?

• Today, we end early for small group feedback... we read every element, and we take it seriously!
Concurrency: Isolation & Consistency

• The DBMS must handle concurrency such that...

1. **Isolation** is maintained: Users must be able to execute each TXN as if they were the only user
   • DBMS handles the details of *interleaving* various TXNs

2. **Consistency** is maintained: TXNs must leave the DB in a consistent state
   • DBMS handles the details of enforcing integrity constraints
Example- consider two TXNs:

T1: START TRANSACTION
UPDATE Accounts
SET Amt = Amt + 100
WHERE Name = ‘A’

UPDATE Accounts
SET Amt = Amt - 100
WHERE Name = ‘B’
COMMIT

T2: START TRANSACTION
UPDATE Accounts
SET Amt = Amt * 1.06
COMMIT

T1 transfers $100 from B’s account to A’s account
T2 credits both accounts with a 6% interest payment
Recall: Three Types of Regions of Memory

1. **Local**: In our model each process in a DBMS has its own local memory, where it stores values that only it “sees”

2. **Global**: Each process can read from / write to shared data in main memory

3. **Disk**: Global memory can read from / flush to disk

4. **Log**: Assume on stable disk storage- spans both main memory and disk...
Scheduling examples

Serial schedule $T_1, T_2$:

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A + = 100$</td>
<td>$B - = 100$</td>
</tr>
<tr>
<td>$A *= 1.06$</td>
<td>$B *= 1.06$</td>
</tr>
</tbody>
</table>

Starting Balance

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$50$</td>
<td>$$200$</td>
</tr>
</tbody>
</table>

Interleaved schedule A:

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A + = 100$</td>
<td>$B - = 100$</td>
</tr>
<tr>
<td>$A *= 1.06$</td>
<td>$B *= 1.06$</td>
</tr>
</tbody>
</table>

Starting Balance

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$159$</td>
<td>$$106$</td>
</tr>
</tbody>
</table>

Same result!
Scheduling examples

Serial schedule $T_1,T_2$:

$T_1$  
A += 100  
B -= 100

$T_2$  
A *= 1.06  
B *= 1.06

Starting Balance

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$50$</td>
<td>$200$</td>
</tr>
</tbody>
</table>

Different result than serial $T_1,T_2$!

Interleaved schedule $B$:

$T_1$  
A += 100  
B -= 100

$T_2$  
A *= 1.06  
B *= 1.06

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$159$</td>
<td>$106$</td>
</tr>
<tr>
<td>$159$</td>
<td>$112$</td>
</tr>
</tbody>
</table>
Scheduling examples

Serial schedule $T_2T_1$:

$T_1$

$A += 100$  $B -= 100$

$T_2$

$A *= 1.06$  $B *= 1.06$

Interleaved schedule B:

$T_1$

$A += 100$  $B -= 100$

$T_2$

$A *= 1.06$  $B *= 1.06$

Starting Balance

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$50</td>
<td>$200</td>
<td></td>
</tr>
</tbody>
</table>

Serial schedule $T_2T_1$:

$A = 153$  $B = 112$

Interleaved schedule B:

$A = 159$  $B = 112$

Different result than serial $T_2T_1$!

ALSO!
Scheduling examples

**Interleaved** schedule B:

T₁: A += 100  
B -= 100

T₂: A *= 1.06  
B *= 1.06

This schedule is different than *any* serial order! We say that it is **not** serializable.
Scheduling Definitions

• A **serial schedule** is one that does not interleave the actions of different transactions

• A and B are **equivalent schedules** if, *for any database state*, the effect on DB of executing A is **identical to** the effect of executing B

• A **serializable schedule** is a schedule that is equivalent to *some* serial execution of the transactions.

The word “some” makes this definition powerful & tricky!
Serializable?

Serial schedules:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1, T_2$</td>
<td>$1.06(A+100)$</td>
<td>$1.06(B-100)$</td>
</tr>
<tr>
<td>$T_2, T_1$</td>
<td>$1.06A + 100$</td>
<td>$1.06B - 100$</td>
</tr>
</tbody>
</table>

Same as a serial schedule for all possible values of $A, B = \text{serializable}$
Serializable?

Serial schedules:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1, T_2$</td>
<td>$1.06(A+100)$</td>
<td>$1.06(B-100)$</td>
</tr>
<tr>
<td>$T_2, T_1$</td>
<td>$1.06A + 100$</td>
<td>$1.06B - 100$</td>
</tr>
</tbody>
</table>

$A += 100$

$B -= 100$

$A *= 1.06$

$B *= 1.06$

Not equivalent to any serializable schedule = not serializable
What else can go wrong with interleaving?

• Various anomalies which break isolation / serializability
  • Often referred to by name...
  • Occur because conflicts between interleaved TXNs
The DBMS’s view of the schedule

An action in the TXNs may
• Reads a value from global memory to local memory
• Write a value from local memory to global memory
• Arbitrary computation on local memory...

Scheduling order matters!
Conflict Types

Two actions **conflict** if they are part of *different* TXNs, involve the same object, and at least one of them is a write.

- Thus, there are three types of conflicts:
  - Read-Write conflicts (RW)
  - Write-Read conflicts (WR)
  - Write-Write conflicts (WW)

Interleaving anomalies occur with / because of these conflicts between TXNs (*but these conflicts can occur without causing anomalies*)!
Classic Anomalies with Interleaved Execution

“Unrepeatable read”: 

Example:

1. T₁ reads some data from A
2. T₂ writes to A
3. Then, T₁ reads from A again and now gets a different / inconsistent value from its own local memory!

Occurring because of a RW conflict. Which pairs?
Occurring because of a WR conflict. Which pairs?

Classic Anomalies with Interleaved Execution

“Dirty read” / Reading uncommitted data:

Example:

$T_1$ writes some data to $A$

$T_2$ reads from $A$, then writes back to $A$ & commits

$T_1$ then aborts - now $T_2$’s result is based on an obsolete / inconsistent value

Occurring because of a WR conflict. Which pairs?
Classic Anomalies with Interleaved Execution

“Inconsistent read” / Reading partial commits:

Example:

1. $T_1$ writes some data to A
2. $T_2$ reads from A and B, and then writes some value which depends on A & B
3. $T_1$ then writes to B - now $T_2$’s result is based on an incomplete commit

Again, occurring because of a WR conflict. Which pairs?
Classic Anomalies with Interleaved Execution

Partially-lost update:

Example:

1. $T_1$ \textit{blind} writes some data to A
2. $T_2$ \textit{blind} writes to A and B
3. $T_1$ then \textit{blind} writes to B; now we have $T_2$'s value for B and $T_1$'s value for A - \textit{not equivalent to any serial schedule}!

Occurring because of a WW conflict. Which pairs?
Activity-8-1.ipynb
2. Conflict Serializability, Locking & Deadlock
What you will learn about in this section

1. RECAP: Concurrency

2. Conflict Serializability

3. DAGs & Topological Orderings

4. Strict 2PL

5. Deadlocks
Recall: Concurrency as Interleaving TXNs

*Serial Schedule:*

\[
T_1: \text{R(A) W(A) R(B) W(B)} \\
T_2: \text{R(A) W(A) R(B) W(B)}
\]

*Interleaved Schedule:*

\[
T_1: \text{R(A) W(A)} \\
T_2: \text{R(A) W(A) R(B) W(B)}
\]

- For our purposes, having TXNs occur concurrently means *interleaving their component actions (R/W)*

We call the particular order of interleaving a **schedule**.
Recall: “Good” vs. “bad” schedules

We want to develop ways of discerning “good” vs. “bad” schedules
Ways of Defining “Good” vs. “Bad” Schedules

• Recall from last time: we call a schedule *serializable* if it is equivalent to *some* serial schedule

  • We used this as a notion of a “good” interleaved schedule, since a *serializable schedule will maintain isolation & consistency*

• Now, we’ll define a stricter, but very useful variant:

  • *Conflict serializability*  
    
    We’ll need to define *conflicts* first..
Conflicts

Two actions **conflict** if they are part of different TXNs, involve the same variable, and at least one of them is a write.
Conflicts

Two actions conflict if they are part of different TXNs, involve the same variable, and at least one of them is a write.

All “conflicts”!
Conflict Serializability

• Two schedules are conflict equivalent if:
  • They involve the same actions of the same TXNs
  • Every pair of conflicting actions of two TXNs are ordered in the same way

• Schedule S is conflict serializable if S is conflict equivalent to some serial schedule

Conflict serializable ⇒ serializable
So if we have conflict serializable, we have consistency & isolation!
Recall: “Good” vs. “bad” schedules

Serial Schedule:

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

Interleaved Schedules:

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

Note that in the “bad” schedule, the order of conflicting actions is different than the above (or any) serial schedule!

Conflict serializability also provides us with an operative notion of “good” vs. “bad” schedules!
Note: Conflicts vs. Anomalies

- **Conflicts** are things we talk about to help us characterize different schedules
  - Present in both “good” and “bad” schedules

- **Anomalies** are instances where isolation and/or consistency is broken because of a “bad” schedule
  - We often characterize different anomaly types by what types of conflicts predicated them
The Conflict Graph

• Let’s now consider looking at conflicts at the TXN level

• Consider a graph where the nodes are TXNs, and there is an edge from $T_i \rightarrow T_j$ if any actions in $T_i$ precede and conflict with any actions in $T_j$
What can we say about “good” vs. “bad” conflict graphs?

Serial Schedule:

Interleaved Schedules:

A bit complicated...
What can we say about “good” vs. “bad” conflict graphs?

**Serial Schedule:**

![Diagram of Serial Schedule]

**Interleaved Schedules:**

![Diagram of Interleaved Schedules]

**Theorem:** Schedule is **conflict serializable** if and only if its conflict graph is **acyclic**
Let’s unpack this notion of acyclic conflict graphs...
DAGs & Topological Orderings

• A topological ordering of a directed graph is a linear ordering of its vertices that respects all the directed edges

• A directed acyclic graph (DAG) always has one or more topological orderings
  • (And there exists a topological ordering if and only if there are no directed cycles)
DAGs & Topological Orderings

• Ex: What is one possible topological ordering here?

Ex: 0, 1, 2, 3 (or: 0, 1, 3, 2)
DAGs & Topological Orderings

• Ex: What is one possible topological ordering here?

There is none!
Connection to conflict serializability

• In the conflict graph, a topological ordering of nodes corresponds to a serial ordering of TXNs

• Thus an acyclic conflict graph $\rightarrow$ conflict serializable!

**Theorem**: Schedule is conflict serializable if and only if its conflict graph is acyclic
Strict Two-Phase Locking

• We consider locking - specifically, strict two-phase locking - as a way to deal with concurrency, because guarantees conflict serializability (if it completes- see upcoming...)

• Also (conceptually) straightforward to implement, and transparent to the user!
Strict Two-phase Locking (Strict 2PL) Protocol:

TXNs obtain:

• An **X** (*exclusive*) lock on object before **writing**.
  - If a TXN holds, no other TXN can get a lock (S or X) on that object.

• An **S** (*shared*) lock on object before **reading**
  - If a TXN holds, no other TXN can get *an X lock* on that object

• All locks held by a TXN are released when TXN completes.
Picture of 2-Phase Locking (2PL)

- # Locks the TXN has:
  - 0 locks

- Lock Acquisition

- Lock Release On TXN commit!

- Time

- Strict 2PL
Strict 2PL

**Theorem:** Strict 2PL allows only schedules whose dependency graph is acyclic

**Proof Intuition:** In strict 2PL, if there is an edge $T_i \rightarrow T_j$ (i.e. $T_i$ and $T_j$ conflict) then $T_j$ needs to wait until $T_i$ is finished – so cannot have an edge $T_j \rightarrow T_i$

Therefore, Strict 2PL only allows conflict serializable $\Rightarrow$ serializable schedules
Strict 2PL

• If a schedule follows strict 2PL and locking, it is conflict serializable...
  • ...and thus serializable
  • ...and thus maintains isolation & consistency!

• Not all serializable schedules are allowed by strict 2PL.

• So let’s use strict 2PL, what could go wrong?
Deadlock Detection: Example

First, $T_1$ requests a shared lock on $A$ to read from it.
Deadlock Detection: Example

Next, $T_2$ requests a shared lock on B to read from it
Deadlock Detection: Example

T₁ then requests an exclusive lock on A to write to it- **now T₂ is waiting on T₁**...
Deadlock Detection: Example

Finally, $T_1$ requests an exclusive lock on $B$ to write to it—now $T_1$ is waiting on $T_2$... DEADLOCK!
ERROR: deadlock detected
DETAIL: Process 321 waits for ExclusiveLock on tuple of relation 20 of database 12002; blocked by process 4924. Process 404 waits for ShareLock on transaction 689; blocked by process 552.
HINT: See server log for query details.

The problem? Deadlock!??!

NB: Also movie called wedlock (deadlock) set in a futuristic prison... I haven’t seen either of them...
Deadlocks

• **Deadlock**: Cycle of transactions waiting for locks to be released by each other.

• Two ways of dealing with deadlocks:
  1. Deadlock prevention
  2. Deadlock detection
Deadlock Detection

• Create the **waits-for graph**:  
  
  • Nodes are transactions  
  
  • There is an edge from $T_i \to T_j$ if $T_i$ is *waiting for $T_j$ to release a lock*  

• Periodically check for *(and break)* cycles in the waits-for graph
Summary

• Concurrency achieved by **interleaving TXNs** such that **isolation & consistency** are maintained
  
  • We formalized a notion of **serializability** that captured such a “good” interleaving schedule

• We defined **conflict serializability**, which implies serializability

• **Locking** allows only conflict serializable schedules
  
  • If the schedule completes... (it may deadlock!)