CPSC 421
Database Management Systems

Lecture 20:
Concurrency Control (cont.)

* Some material adapted from R. Ramakrishnan, L. Delcambre, and B. Ludaescher

Agenda

• Last Assignment (7)
• Concurrency control
Transactions

- A “transaction” is a set of SQL statements (that modify a database) chosen by a user

Transfer $100 from one account to another (generic):

```
BEGIN TRANSACTION
  READ  balance from account 500
  ADD   $100 to balance of account 500
  WRITE new balance to account 500
  READ  balance in account 501
  VERIFY balance to see if it contains at least $100
    ABORT if balance is less than $100
  SUBTRACT $100 from balance of account 501
  WRITE new balance to account 501
COMMIT TRANSACTION
```

this statement is new …
ACID Properties

Atomicity

A transaction happens in its entirety or not at all

- What if the OS crashed half-way through the transaction ... after $100 was added to the first account? (good for the customer!)
- The recovery manager of the DBMS must assure that the $100 is credited back from the first account
- Often called “roll back”
ACID Properties

Consistency

If the DB starts in a consistent state, the transaction will transform it into a consistent state

• The notion of “consistency” is specific to the application constraints (defined by the user)
• Thus the programmer must ensure transactions are consistent
  – The DBMS ensures the transaction is atomic
• E.g., what if the transaction only deposited $100?
  – Probably not consistent according to our example application

Isolation

Each transaction is “isolated” from other transactions … DB state is as if each transaction executed by itself

• What if another transaction computed the total balance after $100 was added to the first account?
• The concurrency control subsystem must
  – ensure that all transactions run in isolation (i.e., don’t mess up other transactions)
  – unless the programmer chooses a less strict level of isolation
  – similar to concurrency control in operating systems
ACID Properties

**Durability**

*If a transaction commits, its changes to the DB state persist (changes are permanent)*

- What if after the commit the OS crashed before the credit was written to disk?
- The *recovery manager* must assure that the credit was at least logged (e.g., to make the DB consistent)

Concurrency

- Why is concurrency important?
  - Better *utilization of resources*
  - E.g., while one user/transaction is reading the disk, another can be using the CPU or reading another disk
  - Results in *better throughput* and *response time*
  - Many applications require it (for performance)
Concurrency

• We’ll look at concurrency in terms of isolation of transactions

  – Isolation is a problem when multiple transactions are running, using the same data, and operations are interleaved

  – Isolation ensured by the concurrency control subsystem

  – This should be familiar if you’ve taken the OS class …

Serial Schedules

• Consider these transactions:
  
  Deposit to A and withdraw from B

  T1: BEGIN A = A + 100; B = B – 100; END

  Compute the balance of A and B

  T2: BEGIN C = A + B; END

  Apply interest to A and B

  T3: BEGIN A = 1.06 * A; B = 1.06 * A; END

  – A schedule is an interleaving of the actions of the transactions so that each transaction’s order is preserved

  – A schedule of transactions is serial if its transactions occur consecutively, one after another
Serial Schedules

• Which of these is a schedule? Which is serial?

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A  =  A + 100</td>
<td>C  =  A + B</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>B  =  B - 100</td>
<td></td>
<td>A  =  1.06 * A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>B  =  1.06 * B</td>
</tr>
<tr>
<td></td>
<td><strong>Serial Schedule!</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A  =  A + 100</td>
<td>C  =  A + B</td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>B  =  B - 100</td>
<td></td>
<td>A  =  1.06 * A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>B  =  1.06 * B</td>
</tr>
<tr>
<td></td>
<td><strong>Non-Serial Schedule!</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A  =  A + 100</td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td>B  =  B - 100</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A  =  1.06 * A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B  =  1.06 * B</td>
</tr>
<tr>
<td></td>
<td><strong>Not a Schedule!</strong></td>
<td></td>
</tr>
</tbody>
</table>

Allowable Concurrency

• What is wrong with S3?
  – It does not give the same result as any serial schedule

• The DBMS should
  – Allow serial schedules like S1
  – Forbid interleaved schedules like S3

• But what about S2?
  – Note that it gives the same result as S1!
  – The DBMS should also allow this schedule!

• If the DBMS only allows serial schedules, then it becomes a batch system (where is the concurrency?)
Serializable Schedules

• A schedule is “serializable” if its effect on the DB is the same as the effect on some serial schedule
  – Serial schedules are always serializable
  – S2 is serializable, but S3 is not
• Serializability is the same as the isolation condition
• The goal of the concurrency control subsystem is to ensure serializability

Schedules as Reads and Writes

• An expression A = 1.06 * A means
  – Read A from disk
  – Set A equal to 1.06 * A
  – Write A to disk
• Only the read and write to disk matter to the DBMS!
  – We’ll use the notation …
  – R(A) for Read A
  – W(A) for Write A
Schedules as Reads and Writes

- These are equal (from DBMS/concurrency perspective)

<table>
<thead>
<tr>
<th>S2</th>
<th>T1</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A = A + 100</td>
<td>A = 1.06 * A</td>
</tr>
<tr>
<td></td>
<td>B = B - 100</td>
<td>B = 1.06 * B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S2</th>
<th>T1</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R(A), W(A)</td>
<td>R(A), W(A)</td>
</tr>
<tr>
<td></td>
<td>R(B), W(B)</td>
<td>R(B), W(B)</td>
</tr>
</tbody>
</table>

Conflict Serializability

- S2 has a special structure that makes it possible to show that it is serializable …

- Two actions are “nonconflicting” if they are in different transactions and either they
  - Access different data times (resources)
  - Or both are reads

- If nonconflicting actions are commuted then the new schedule gives the same result
Conflict Serializability

• Two schedules are “conflict equivalent” if
  – One can be transformed into the other by commuting (swapping) nonconflicting actions

• A schedule is “conflict serializable” if it is conflict equivalent to at least one serial schedule

Thus, every conflict serializable schedule is serializable!
(... but not nec. the other way around)

Serializability

• Nonconflicting shown in Red

<table>
<thead>
<tr>
<th>Schedule S2</th>
<th>T1</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A), W(A)</td>
<td>R(B), W(B)</td>
<td>R(B), W(B)</td>
</tr>
<tr>
<td>R(B), W(B)</td>
<td>R(A), W(A)</td>
<td>R(A), W(A)</td>
</tr>
</tbody>
</table>

A is used in T3, B in T1

Commute

This is a serial schedule!
This means S2 is “conflict serializable” (... and thus is serializable)
**Precedence Graphs**

- Verifying conflict serializability is tedious
- There is an easier way!

**Precedence graphs**

- One node per transaction
- Edge from $T_i$ to $T_j$ if an action in $T_i$ occurs before an action in $T_j$ and the actions conflict

**Theorem**

- A schedule is **conflict serializable** if and only if its precedence graph is **acyclic**

---

**Exercise**

- With a partner, draw the precedence graphs for the previous schedules (S1-S3)
  - You'll first have to convert them to R's and W's

- **S1**
  - T1: R(A), W(A), R(B), W(B)
  - T2: R(A), R(B), W(B)
  - **Serial!**

- **S2**
  - T1: R(A), W(A), R(B), W(B)
  - T2: R(A), W(A), W(B)
  - **Not Serial!**

- **S3**
  - T1: R(A), W(A), R(B), W(B)
  - T2: R(A), R(B), W(C)
  - **Not Serial!**
Exercise

• With a partner, draw the precedence graphs for these schedules

<table>
<thead>
<tr>
<th>S5</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T2</td>
<td>T3</td>
</tr>
<tr>
<td>R(A)</td>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>R(B)</td>
<td>W(A)</td>
<td>R(B)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S6</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T2</td>
<td>T3</td>
</tr>
<tr>
<td>R(A)</td>
<td>W(A)</td>
<td>W(A)</td>
</tr>
</tbody>
</table>

S6 is actually serializable: whichever transaction writes A last "wins"
Serializable but not Conflict Serializable

Relationships ...

Serializable
Conflict Serializable
Serial
Acyclic Precedence Graph
**Serializability in Practice …**

- Precedence graphs give us a simple way to prove that a schedule is (conflict) serializable
  - But they do not work for all schedules (e.g., S9)
  - In theory, this could be used by a DBMS to check for serializability …

**In practice**

- A DBMS is not presented with schedules … it only sees a stream of transactions
  - Instead, locking is used to achieve “isolation”

---

**Locking**

- Transactions **must obtain a lock** before reading or updating (writing) data

- Two kinds of locks:
  - Shared (S) locks
  - Exclusive (X) locks

- To **read a record** you MUST get an S lock
- To **write (modify/delete) a record** you MUST get an X lock

- Lock information is maintained by a “lock manager”
How Locks Work

• If an object has an S (shared) lock
  – new transactions can obtain S (shared) locks
  – but *not* X (exclusive) locks

• If an object has an X lock
  – no other transaction can obtain any lock on that object

• If a transaction cannot obtain a lock
  – It is *blocked* (i.e., waits in a queue)

Strict Two Phase Locking Protocol (*Strict 2PL*)

• In Strict 2PL
  – Transaction T obtains (S and X) locks gradually, as needed
  – T holds all locks until end of transaction (commit/abort)
Strict Two Phase Locking Protocol (Strict 2PL)

- In Strict 2PL
  - Transaction T obtains (S and X) locks gradually, as needed
  - T holds all locks until end of transaction (commit/abort)

  This guarantees serializability!
  - Still permits interleaved schedules
  - But can lead to deadlock … :-(

Strict Two Phase Locking Protocol (Strict 2PL)

- Examples ...

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>S6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X(B)</td>
<td>X(A)</td>
</tr>
<tr>
<td>W(B)</td>
<td>W(A)</td>
</tr>
<tr>
<td>X(C)</td>
<td>X(C)</td>
</tr>
<tr>
<td>W(C)</td>
<td>W(C)</td>
</tr>
<tr>
<td>Commit</td>
<td>Commit</td>
</tr>
<tr>
<td>S7 (S2PL)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>X(A)</td>
<td>R(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>Commit</td>
<td>Commit</td>
<td>Commit</td>
</tr>
<tr>
<td>S6 (S2PL)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Strict Two Phase Locking Protocol (*Strict 2PL*)

- What about this one?
  - T1: \( W(A), W(B) \)
  - T2: \( W(B), W(A) \)

- Oops … T1 and T2 are “deadlocked”

<table>
<thead>
<tr>
<th>S8 (S2PL)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T1</strong></td>
</tr>
<tr>
<td>X(A)</td>
</tr>
<tr>
<td>W(A)</td>
</tr>
<tr>
<td>Waiting for X(B)</td>
</tr>
</tbody>
</table>