Database Applications (15-415)

DBMS Internals- Part XI
Lecture 22, April 12, 2015

Mohammad Hammoud
Today...

- **Last Session:**
  - DBMS Internals- Part X
    - Query Optimization (*Cont’d*)

- **Today’s Session:**
  - DBMS Internals- Part XI
    - Transaction Management

- **Announcements:**
  - The grades of Quiz II are out
  - PS4 is due today by midnight
  - PS5 will be posted on Tuesday. It is due on Thursday, April 23rd
DBMS Layers

Queries

Query Optimization and Execution

Relational Operators

Files and Access Methods

Buffer Management

Disk Space Management

Transaction Manager

Lock Manager

Recovery Manager

DB
Outline

A Brief Primer on Transaction Management

Anomalies Due to Concurrency

2PL and Strict 2PL Locking Protocols

Schedules with Aborted Transactions
Concurrent Execution of Programs

- A database is typically *shared* by a large number of users.

- DBMSs *schedule* users’ programs *concurrently*:
  - While one user program is waiting for an I/O access to be satisfied, the CPU can process another program:
    - Better system throughput
  - Interleaved execution of a short program with a long program allows the short program to complete quickly:
    - Better response time
    - Better for fairness reasons
Transactions

- *Any one* execution of a user program in a DBMS is denoted as a **transaction**
  - Executing the same program several times will generate several transactions

- A transaction is the basic unit of change as seen by a DBMS
  - E.g., Transfer $100 from account A to account B

- A transaction may carry out many operations on data, but DBMSs are only concerned about *reads* and *writes*

- Thus, in essence a transaction becomes *a sequence of reads and writes*
Transactions (*Cont’d*)

- In addition to reading and writing, a transaction must specify as its final action:
  - Either *Commit* (i.e., complete successfully)
  - Or *Abort* (i.e., terminate and *undo* actions)

- We make two assumptions:
  - Transactions interact only via database reads and writes (i.e., no *message passing*)
  - A database is a fixed collection of *independent* objects (A, B, C, etc.)
Schedules

- A **schedule** is a list of actions (i.e., read, write, abort, and/or commit) from a *set* of transactions.

- The *order* in which two actions of a transaction $T$ appear in a schedule must be the same as they appear in $T$ itself.

- Assume $T_1 = [R(A), W(A)]$ and $T_2 = [R(B), W(B), R(C), W(C)]$.

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

- ✅  
- ✅  
- ✗
Serial Schedules

- A **complete schedule** must contain all the actions of every transaction that appears on it.

- If the actions of different transactions are *not interleaved*, the schedule is called a **serial schedule**.

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

A Serial Schedule

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

A Non-Serial Schedule
Serializable Schedules

- Two schedules are said to be *equivalent* if for any database state, the effect of executing the 1st schedule is identical to the effect of executing the 2nd schedule.

- A *serializable schedule* is a schedule that is equivalent to a serial schedule.

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

A *Serializable Schedule*

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

A *Serial Schedule*

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

Another *Serializable Schedule*
Examples

- Assume transactions T1 and T2 as follows:

  T1: \[
  \text{BEGIN} \quad A = A - 100, \quad B = B + 100 \quad \text{END}
  \]

  T2: \[
  \text{BEGIN} \quad A = 1.06 \times A, \quad B = 1.06 \times B \quad \text{END}
  \]

- T1 can be thought of as transferring $100 from A’s account to B’s account.

- T2 can be thought of as crediting accounts A and B with a 6% interest payment.
Examples: A *Serial* Schedule

- Assume transactions T1 and T2 as follows:

T1:    BEGIN   A=A-100,   B=B +100   END
T2:    BEGIN   A=1.06*A,   B=1.06*B   END

*T1: Transfer $100 from A to B*

1. Bal=960
2. Bal=960
3. 3

*T2 = Add interest of 6% to A and B*

4. Bal=1160
5. 1

Account A

Account B
Examples: Another *Serial* Schedule

- Assume transactions T1 and T2 as follows:

  T1: \[\text{BEGIN } A = A - 100, \ B = B + 100 \ \text{END}\]
  T2: \[\text{BEGIN } A = 1.06 \times A, \ B = 1.06 \times B \ \text{END}\]

**Account A**
- Bal = 954

**Account B**
- Bal = 1166

*Previously:*
- Account A = 960
- Account B = 1160

*T1: Transfer $100 from A to B*

*T2 = Add interest of 6% to A and B*
Examples: A *Serializable* Schedule

- Assume transactions T1 and T2 as follows:

  T1: \[ \text{BEGIN} \ A = A - 100, \ B = B + 100 \ \text{END} \]
  T2: \[ \text{BEGIN} \ A = 1.06 \times A, \ B = 1.06 \times B \ \text{END} \]

### Previous Serial Schedule:

- **Account A**: 954
- **Account B**: 1166

**T1**: Transfer $100 from A to B

1. **Bal = 954**
2. **Bal = 900**
3. **Bal = 954**
4. **Bal = 1166**

**T2**: Add interest of 6% to A and B
Comments

- 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.

- Executing transactions serially in different orders may produce different results, but they are all acceptable!

- The DBMS makes no guarantees about which result will be the outcome of an interleaved execution.
Outline

A Brief Primer on Transaction Management

Anomalies Due to Concurrency

2PL and Strict 2PL Locking Protocols

Schedules with Aborted Transactions
Anomalies

- Interleaving actions of different transactions can leave the database in an **inconsistent state**

- Two actions on the same data object are said to **conflict** if at least one of them is a write

- There are 3 **anomalies** that can arise upon interleaving actions of different transactions (say, T1 and T2):
  - **Write-Read (WR) Conflict**: T2 reads a data object previously written by T1
  - **Read-Write (RW) Conflict**: T2 writes a data object previously read by T1
  - **Write-Write (WW) Conflict**: T2 writes a data object previously written by T1
WR conflicts arise when transaction T2 reads a data object A that has been modified by another transaction T1, which has not yet committed. Such a read is called a dirty read.

Assume T1 and T2 such that:

- T1 transfers $100 from A’s account to B’s account
- T2 credits accounts A and B with a 6% interest payment

```
T1: BEGIN  A=A-100,  B=B +100  END
T2: BEGIN  A=1.06*A,  B=1.06*B  END
```
Suppose that T1 and T2 actions are *interleaved* as follows:

- T1 deducts $100 from account A
- T2 adds 6% interest to accounts A and B
- T1 credits $100 to account B

**T1: Transfer $100 from A to B**

**T2 = Add interest of 6% to A and B**

Account A

Bal=1000

Account B

Bal=1000
Suppose that T1 and T2 actions are *interleaved* as follows:

1. T1 deducts $100 from account A
2. T2 adds 6% interest to accounts A and B
3. T1 credits $100 to account B

### Diagram

- **T1:** Transfer $100 from A to B
- **T2:** Add interest of 6% to A and B

#### Account A
- Bal = 954
- Bal = 1054

#### Account B
- Bal = 1160
- Bal = 1160

*Different than any serial schedule.* (I.e., Neither: [A = 954 and B = 1166] Nor: [A = 960 and B = 1160])
Reading Uncommitted Data: WR Conflicts

- T1 and T2 can be represented by the following schedule:

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

The value of A written by T1 is read by T2 before T1 has completed all its changes!

Why is this a problem?
Reading Uncommitted Data: WR Conflicts

- T1 and T2 can be represented by the following schedule:

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

The value of A written by T1 is read by T2 before T1 has completed all its changes!

Why is this a problem?

- T1 may write some value into A that makes the database inconsistent.
- As long as T1 overwrites this value with a ‘correct’ value of A before committing, no harm is done if T1 and T2 are run in some serial order (this is because T2 would then not see the temporary inconsistency).
Reading Uncommitted Data: WR Conflicts

- T1 and T2 can be represented by the following schedule:

  T1                      T2
  R(A)  W(A)              R(A)  W(A)
  R(B)  W(B)              R(B)  W(B)
  Commit                   Commit

  The value of A written by T1 is read by T2 before T1 has completed all its changes!

  Why is this a problem?

Note that although a transaction must leave a database in a consistent state after it completes, it is not required to keep the database consistent while it is still in progress!
Unrepeatable Reads: RW Conflicts

- RW conflicts arise when transaction T2 writes a data object A that has been read by another transaction T1, while T1 is still in progress

- If T1 tries to read A again, it will get a different result!
  - Such a read is called an unrepeatable read

- Assume A is the number of available copies for a book
  - A transaction that places an order on the book reads A, checks that A > 0 and decrements A
  - Assume two transactions, T1 and T2
Unrepeatable Reads: RW Conflicts

- Suppose that T1 and T2 actions are interleaved as follows:
  - T1 reads A
  - T2 reads A, decrements A and commit
  - T1 tries to decrement A

This situation will never arise in a serial execution of T1 and T2; T2 would read A and see 0 and therefore not proceed with placing an order!
Overwriting Uncommitted Data: WW Conflicts

- WW conflicts arise when transaction T2 writes a data object A that has been written by another transaction T1, while T1 is still in progress.

- Suppose that Mohammad and Ahmad are two employees and their salaries must be kept equal.

- Assume T1 sets Mohammad’s and Ahmad’s salaries to $1000.

- Assume T2 sets Mohammad’s and Ahmad’s salaries to $2000.
Overwriting Uncommitted Data: WW Conflicts

**T1: Sets Salaries to $1000**

**T2 = Sets Salaries to $2000**

Mohammad’s Salary

- MS = 1000
- AS = 2000

Ahmad’s Salary

- MS = 1000
- AS = 1000

Diagram:

1. Mohammad’s Salary
2. MS = 1000
3. Mohammad’s Salary
4. AS = 1000

Conflicts resolved.
Overwriting Uncommitted Data: WW Conflicts

Either serial schedule is acceptable from a consistency standpoint (although Mohammad and Ahmad may prefer a higher salary!)

Neither T1 nor T2 reads a salary value before writing it—such a write is called a blind write!
Overwriting Uncommitted Data: WW Conflicts

The problem is that we have a **lost update**. In particular, T2 overwrote Mohammad’s Salary as set by T1 (this will never happen with a serializable schedule!)
Outline

A Brief Primer on Transaction Management

Anomalies Due to Concurrency

2PL and Strict 2PL Locking Protocols

Schedules with Aborted Transactions
Locking Protocols

- WR, RW and WW anomalies can be avoided using a locking protocol

- A locking protocol:
  - Is a set of rules to be followed by each transaction to ensure that only serializable schedules are allowed (extended later)
  - Associates a lock with each database object, which could be of different types (e.g., shared or exclusive)
  - Grants and denies locks to transactions according to the specified rules

- The part of the DBMS that keeps track of locks is called the lock manager
Lock Managers

- Usually, a lock manager in a DBMS maintains three types of data structures:
  - A queue, $Q$, for each lock, $L$, to hold its pending requests
  - A lock table, which keeps for each $L$ associated with each object, $O$, a record $R$ that contains:
    - The type of $L$ (e.g., shared or exclusive)
    - The number of transactions currently holding $L$ on $O$
    - A pointer to $Q$
  - A transaction table, which maintains for each transaction, $T$, a pointer to a list of locks held by $T
Two-Phase Locking

- A widely used locking protocol, called **Two-Phase Locking (2PL)**, has two rules:
  - **Rule 1**: if a transaction \( T \) wants to read (or write) an object \( O \), it first requests the lock manager for a shared (or exclusive) lock on \( O \)
Two-Phase Locking

- A widely used locking protocol, called **Two-Phase Locking (2PL)**, has two rules:
  - **Rule 1**: if a transaction $T$ wants to read (or write) an object $O$, it first requests the lock manager for a shared (or exclusive) lock on $O$. 

![Diagram of Two-Phase Locking](image)
Two-Phase Locking

- A widely used locking protocol, called *Two-Phase Locking* (*2PL*), has two rules:
  - **Rule 1**: if a transaction $T$ wants to read (or write) an object $O$, it first requests the lock manager for a shared (or exclusive) lock on $O$. 

![Diagram of Two-Phase Locking](image)
Two-Phase Locking

- A widely used locking protocol, called *Two-Phase Locking (2PL)*, has two rules:
  - **Rule 1**: if a transaction $T$ wants to read (or write) an object $O$, it first requests the lock manager for a shared (or exclusive) lock on $O$.
Two-Phase Locking

- A widely used locking protocol, called \textit{Two-Phase Locking (2PL)}, has two rules:
  - Rule 2: \(T\) can release locks before it \textit{commits} or \textit{aborts}, and cannot request additional locks once it releases \textit{any} lock.

- Thus, every transaction has a “growing” phase in which it acquires locks, followed by a “shrinking” phase in which it releases locks.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{Figures/two_phase_locking}
\caption{Graphical representation of the growing and shrinking phases of a transaction using Two-Phase Locking.}
\end{figure}
Two-Phase Locking

- A widely used locking protocol, called \textit{Two-Phase Locking (2PL)}, has two rules:
  - Rule 2: \textit{T} can release locks before it \textit{commits} or \textit{aborts}, and cannot request additional locks once it releases \textit{any} lock.

- Thus, every transaction has a “growing” phase in which it acquires locks, followed by a “shrinking” phase in which it releases locks.

\begin{center}
\includegraphics[width=\textwidth]{violation_of_2pl.png}
\end{center}
Resolving RW Conflicts Using 2PL

- Suppose that T1 and T2 actions are interleaved as follows:
  - T1 reads A
  - T2 reads A, decrements A and commit
  - T1 tries to decrement A

- T1 and T2 can be represented by the following schedule:
Resolving RW Conflicts Using 2PL

- Suppose that T1 and T2 actions are interleaved as follows:
  - T1 reads A
  - T2 reads A, decrements A and commit
  - T1 tries to decrement A

- T1 and T2 can be represented by the following schedule:

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

- Exposes RW Anomaly

But, it can limit parallelism!
Resolving WW Conflicts Using 2PL

- Suppose that T1 and T2 actions are interleaved as follows:
  - T1 sets Mohammad’s Salary to $1000
  - T2 sets Ahmad’s Salary to $2000
  - T1 sets Ahmad’s Salary to $1000
  - T2 sets Mohammad’s Salary to $2000

- T1 and T2 can be represented by the following schedule:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>W(MS)</td>
<td>W(AS)</td>
<td>EXCLUSIVE(MS)</td>
<td>EXCLUSIVE(AS)</td>
</tr>
<tr>
<td>W(AS)</td>
<td>W(MS)</td>
<td>W(MS)</td>
<td>W(AS)</td>
</tr>
<tr>
<td>Commit</td>
<td>Commit</td>
<td>Commit</td>
<td>Commit</td>
</tr>
</tbody>
</table>

Exposes WW Anomaly (assuming, MS & AS must be kept equal)

WW Conflict Resolved!
Resolving WW Conflicts Using 2PL

- Suppose that T1 and T2 actions are interleaved as follows:
  - T1 sets Mohammad’s Salary to $1000
  - T2 sets Ahmad’s Salary to $2000
  - T1 sets Ahmad’s Salary to $1000
  - T2 sets Mohammad’s Salary to $2000

- T1 and T2 can be represented by the following schedule:

```
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>W(MS)</td>
<td>W(AS)</td>
</tr>
<tr>
<td>W(AS)</td>
<td>Lock(AS)</td>
</tr>
<tr>
<td>Commit</td>
<td>W(MS)</td>
</tr>
<tr>
<td></td>
<td>Lock(MS)</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>
```

Exposes WW Anomaly
(assuming, MS & AS must be kept equal)

Deadlock!
Resolving WR Conflicts

- Suppose that T1 and T2 actions are *interleaved* as follows:
  - T1 deducts $100 from account A
  - T2 adds 6% interest to accounts A and B
  - T1 credits $100 to account B

- T1 and T2 can be represented by the following schedule:

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

Exposes WR Anomaly

**WR Conflict Resolved!**
Resolving WR Conflicts

- Suppose that T1 and T2 actions are *interleaved* as follows:
  - T1 deducts $100 from account A
  - T2 adds 6% interest to accounts A and B
  - T1 credits $100 to account B

- T1 and T2 can be represented by the following schedule:

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

How can we solve this?
Strict Two-Phase Locking

- WR conflicts (as well as RW & WW) can be solved by making 2PL stricter

- In particular, Rule 2 in 2PL can be modified as follows:
  - Rule 2: locks of a transaction $T$ can only be released after $T$ completes (i.e., commits or aborts)

- This version of 2PL is called *Strict Two-Phase Locking*
Resolving WR Conflicts: *Revisit*

- Suppose that T1 and T2 actions are *interleaved* as follows:
  - T1 deducts $100 from account A
  - T2 adds 6% interest to accounts A and B
  - T1 credits $100 to account B

- T1 and T2 can be represented by the following schedule:

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

Exposes WR Anomaly
```

```
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EXCLUSIVE(A)</td>
</tr>
<tr>
<td></td>
<td>EXCLUSIVE(B)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Lock(A)</td>
<td>EXCLUSIVE(A)</td>
</tr>
<tr>
<td>Lock(B)</td>
<td>EXCLUSIVE(B)</td>
</tr>
<tr>
<td>Wait</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RELEASE(A)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

Not allowed with *strict 2PL*
```
Resolving WR Conflicts: *Revisit*

- Suppose that T1 and T2 actions are *interleaved* as follows:
  - T1 deducts $100 from account A
  - T2 adds 6% interest to accounts A and B
  - T1 credits $100 to account B

- T1 and T2 can be represented by the following schedule:

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

Exposes WR Anomaly

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

WR Conflict is Resolved!

But, parallelism is limited more!
2PL vs. Strict 2PL

- **Two-Phase Locking (2PL):**
  - Limits concurrency
  - May lead to deadlocks
  - May have ‘dirty reads’

- **Strict 2PL:**
  - Limits concurrency more *but, actions of different transactions can still be interleaved*
  - May still lead to deadlocks
  - Avoids ‘dirty reads’

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

A Schedule with *Strict 2PL* and *Interleaved Actions*
Performance of Locking

- Locking comes with delays mainly from *blocking*.

- Usually, the first few transactions are unlikely to conflict.
  - Throughput can rise in proportion to the number of active transactions.

- As more transactions are executed concurrently, the likelihood of blocking increases.
  - Throughput will increase more slowly with the number of active transactions.

- There comes a point when adding another active transaction will actually decrease throughput.
  - When the system *thrashes*!
If a database begins to *thrash*, the DBA should reduce the number of active transactions.

Empirically, thrashing is seen to occur when 30% of active transactions are blocked!
Outline

A Brief Primer on Transaction Management

Anomalies Due to Concurrency

2PL and Strict 2PL Locking Protocols

Schedules with Aborted Transactions
Schedules with *Aborted* Transactions

- Suppose that T1 and T2 actions are interleaved as follows:
  - T1 deducts $100 from account A
  - T2 adds 6% interest to accounts A and B, and commits
  - T1 is aborted

- T1 and T2 can be represented by the following schedule:

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

  **T2 read a value for A that should have never been there!**

  **How can we deal with the situation, assuming T2 *had not yet committed*?**
Schedules with *Aborted* Transactions

- Suppose that T1 and T2 actions are interleaved as follows:
  - T1 deducts $100 from account A
  - T2 adds 6% interest to accounts A and B, and commits
  - T1 is aborted

- T1 and T2 can be represented by the following schedule:

  T1
  \[
  \begin{array}{c|c}
  T1 & T2 \\
  \hline
  R(A) & R(A) \\
  W(A) & W(A) \\
  \end{array}
  \]

  T2
  \[
  \begin{array}{c|c}
  & R(B) \\
  \hline
  W(B) & \text{Commit} \\
  \end{array}
  \]

  We can *cascade* the abort of T1 by aborting T2 as well!

  T2 read a value for A that should have never been there!

  This “cascading process” can be *recursively* applied to any transaction that read A written by T1.
Schedules with *Aborted* Transactions

- Suppose that T1 and T2 actions are interleaved as follows:
  - T1 deducts $100 from account A
  - T2 adds 6% interest to accounts A and B, and commits
  - T1 is aborted

- T1 and T2 can be represented by the following schedule:

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

T2 read a value for A that should have never been there!

How can we deal with the situation, assuming T2 *had actually committed*?

The schedule is indeed *unrecoverable*!
Schedules with *Aborted* Transactions

- Suppose that T1 and T2 actions are interleaved as follows:
  - T1 deducts $100 from account A
  - T2 adds 6% interest to accounts A and B, and commits
  - T1 is aborted

- T1 and T2 can be represented by the following schedule:

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

T2 read a value for A that should have never been there!

For a schedule to be *recoverable*, transactions should commit only after all transactions whose changes they read commit!

“*Recoverable schedules*” avoid *cascading aborts*!
Schedules with *Aborted* Transactions

- Suppose that T1 and T2 actions are interleaved as follows:
  - T1 deducts $100 from account A
  - T2 adds 6% interest to accounts A and B, and commits
  - T1 is aborted

- T1 and T2 can be represented by the following schedule:

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

How can we ensure “recoverable schedules”?

By using Strict 2PL!
Schedules with *Aborted* Transactions

- Suppose that T1 and T2 actions are interleaved as follows:
  - T1 deducts $100 from account A
  - T2 adds 6% interest to accounts A and B, and commits
  - T1 is aborted

- T1 and T2 can be represented by the following schedule:

```plaintext
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td>EXCLUSIVE(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>Abort UNDO(T1)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td>Lock(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>Wait</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td>EXCLUSIVE(A)</td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td></td>
</tr>
</tbody>
</table>
```

Cascaded aborts are avoided!
Serializable Schedules: Redefined

- Two schedules are said to be *equivalent* if for any database state, the effect of executing the 1st schedule is identical to the effect of executing the 2nd schedule.

- Previously: a *serializable schedule* is a schedule that is equivalent to a serial schedule.

- Now: a *serializable schedule* is a schedule that is equivalent to a serial schedule over a set of committed transactions.

- This definition captures *serializability* as well as *recoverability*.
Next Class

Queries

- Query Optimization and Execution
- Relational Operators
- Files and Access Methods
- Buffer Management
- Disk Space Management

Database (DB)

Transaction Manager

Lock Manager

Recovery Manager

Continue…