Concurrency

By Marina Barsky
Winter 2016, University of Toronto
Database Management Systems

• Where and how data is stored
• How to process large amounts of data efficiently
• How to ask questions about data— and how to implement efficient answers to these questions
• **How to preserve data integrity**: dealing with crashes and concurrency
Why concurrent execution

• In production environments, it is unlikely that we can limit our system to just one user at a time.
  • Consequently, it is possible for multiple queries to be submitted at approximately the same time.

• If all of the queries were very small (i.e., in terms of time), we could probably just execute them serially, on a first-come-first-served basis.

• However, many queries are both complex and time consuming.
  • Executing these queries would make other queries wait a long time for a chance to execute.
  • Disk usage can be optimized for several queries running in parallel (recall – elevator algorithm)

• So, in practice, the DBMS may be running many different queries at about the same time.
Problems with concurrency: two people—one bank account

- Before withdrawing money, each needs to check if the balance is sufficient
- Initially there is 100$ on the account

<table>
<thead>
<tr>
<th>Ryan</th>
<th>Monica</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(X, b)</td>
<td>READ(X, c)</td>
</tr>
<tr>
<td>b = 100</td>
<td>c = 100</td>
</tr>
<tr>
<td>b - = 100</td>
<td>c - = 50</td>
</tr>
<tr>
<td>WRITE (X, Ryan)</td>
<td>WRITE (X, c)</td>
</tr>
</tbody>
</table>

Ryan: thinks 0 $ left
Monica: thinks 50 $ left

In fact, the withdrawn amount is 150$
Problems with concurrency: two people—one bank account

- Before withdrawing money, each needs to check if the balance is sufficient
- Initially there is 100$ on the account

<table>
<thead>
<tr>
<th>Ryan</th>
<th>Monica</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(X, b)</td>
<td></td>
</tr>
<tr>
<td>b = 100</td>
<td></td>
</tr>
<tr>
<td>WRITE(X, b)</td>
<td></td>
</tr>
</tbody>
</table>

The problem is that the reading and writing operations should be performed as one **transaction**, their combination should be **atomic**
Transaction

• DBMS groups your SQL statements into **transactions**.
• The **transaction** is the atomic unit of execution of database operations.
• By default, each query or DML statement is a transaction.
• With embedded SQL user can group multiple SQL statements into a single transaction.
• The transaction ends when one of the following occurs:
  • A **COMMIT** or **ROLLBACK** are issued.
  • A DDL (CREATE, ALTER, DROP ...) or DCL (GRANT, REVOKE) statement is issued.
  • A user properly exits (COMMIT).
  • System crashes (ROLLBACK).

Subsets of SQL

Queries: SELECT
Data Manipulation Language (DML): INSERT, UPDATE, DELETE
Data Definition Language (DDL): CREATE, ALTER, DROP, RENAME
Transaction control: COMMIT, ROLLBACK
Data Control Language (DCL): GRANT, REVOKE
AUTOCOMMIT (Oracle syntax)

• Environment variable **AUTOCOMMIT** is by default set to OFF
• A user can set it to
  
  SET AUTOCOMMIT ON
  
  SET AUTOCOMMIT IMMEDIATE
  
  and every SQL statement becomes a single transaction

• In the middle of transaction, the user can see the changed data by issuing SELECT queries.
• The user is getting the data from the temporary storage area.
• Other users cannot see the changes until transaction has been committed
Transaction properties: **ACID**

- **Atomicity**: Whole transaction or none is done.
- **Consistency**: Database constraints preserved. Transaction, executed completely, takes database from one *consistent state* to another.
- **Isolation**: It appears to the user as if only one process executes at a time.
  - That is, even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions one after another in some *serial order*.
- **Durability**: Effects of a process survive a crash.
Interleaving

• For performance reasons, a DBMS has to interleave the actions of several transactions.

• Interleaving of transactions may lead to anomalies even if each individual transaction preserves all the database constraints.
Transactions and Schedules: notation

• To reason about the order of interleaving transactions, we can abstract each transaction into a sequence of reads and writes of disk data.

• For example, withdrawing of money from the account can be written as:

  \( r_1(A); w_1(A) \)

That means that transaction \( T_1 \) reads database element \( A \), does something with it in main memory and writes it back to the database.

• Then we can record the sequence of commands received by DBMS as:

  \( r_1(A); w_1(A); r_2(A); w_2(A) \)
Transactions and Schedules: notation

• To ensure that interleaving does not lead to anomalies, DBMS schedules the execution of each action in a certain way

• A **schedule** is a list of actions for a set of interleaved transactions

• For example:

  T1__________________T2
  r(A)
  r(A)
  r(A)
  w(A)
  w(A)
  commit
  commit
Anomalies of interleaving transactions: example 1

• Consider two transactions T1 and T2, each of which, when running alone preserves database consistency:
  • T1 transfers $100 from A to B (e.g. from checking to saving account)
  • T2 increments both A and B by 1% (e.g. daily interest)

• The list of actions received by DBMS:
  \[
  r_1(A); \ w_1(A);r_1(B);w_1(B);r_2(A);w_2(A);r_2(B);w_2(B)
  \]
Anomalies of interleaving transactions: example 1

DBMS decides on the following schedule:

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(A)</td>
<td></td>
<td>r(A)</td>
</tr>
<tr>
<td>w(A)</td>
<td>r(A)</td>
<td>w(A)</td>
</tr>
<tr>
<td>r(B)</td>
<td>w(B)</td>
<td>commit</td>
</tr>
<tr>
<td>w(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>commit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What is the problem?
Anomalies of interleaving transactions: example 1

T1 r(A) w(A) T2
r(A) w(A) T1 deducted $100 from A
r(A) w(A) T2 incremented both A and B by 1%
r(B) commit T1 added $100 to B
r(B) w(B) commit

Anomalies: reading uncommitted data

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(A)</td>
<td>T1 deducted $100 from A</td>
</tr>
<tr>
<td>w(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T2 incremented both A and B</td>
</tr>
<tr>
<td></td>
<td>by 1%</td>
</tr>
<tr>
<td>r(B)</td>
<td>T1 added $100 to B</td>
</tr>
<tr>
<td>w(B)</td>
<td></td>
</tr>
<tr>
<td>commit</td>
<td></td>
</tr>
<tr>
<td>r(B)</td>
<td></td>
</tr>
<tr>
<td>w(B)</td>
<td></td>
</tr>
<tr>
<td>commit</td>
<td></td>
</tr>
</tbody>
</table>

The problem is that the bank didn’t pay interest on the $100 that was being transferred. This happened because T2 was reading uncommitted values.
Anomalies of interleaving transactions: example 2

- Suppose that $A$ is the number of copies available for a book.

- Transactions $T_1$ and $T_2$ both place an order for this book. First they check the availability of the book.

- Consider the following scenario:
  1. $T_1$ checks whether $A$ is greater than 1.  
     Suppose $T_1$ sees (reads) value 1.
  2. $T_2$ also reads $A$ and sees 1.
  3. $T_2$ decrements $A$ to 0.
  4. $T_2$ commits.
  5. $T_1$ tries to decrement $A$, which is now 0, and gets an error because some integrity check doesn’t allow it.
Anomalies: 
unrepeatable reads

1. T1 checks whether A is greater than 1. 
   Suppose T1 sees (reads) value 1. 
2. T2 also reads A and sees 1. 
3. T2 decrements A to 0. 
4. T2 commits. 
5. T1 tries to decrement A, which is now 0, and gets an error because some integrity check doesn’t allow it.

The problem is that because value of A has been changed by T1, when T2 reads A for the second time, before updating it, the value is different from that when T2 started.
Anomalies of interleaving transactions: example 3

- Suppose that Larry and Harry are two employees, and their salaries must be kept equal. T1 sets their salaries to $1000 and T2 sets their salaries to $2000.

- Now consider the following schedule:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(Larry)</td>
<td>r(Harry)</td>
</tr>
<tr>
<td>w(Larry)</td>
<td>w(Harry)</td>
</tr>
<tr>
<td>r(Harry)</td>
<td>r(Larry)</td>
</tr>
<tr>
<td>w(Harry)</td>
<td>w(Larry)</td>
</tr>
<tr>
<td></td>
<td>commit</td>
</tr>
</tbody>
</table>

What is the problem?
Suppose that Larry and Harry are two employees, and their salaries must be kept equal. T1 sets their salaries to $1000 and T2 sets their salaries to $2000.

Now consider the following schedule:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(Larry)</td>
<td></td>
</tr>
<tr>
<td>w(Larry)</td>
<td></td>
</tr>
<tr>
<td>r(Harry)</td>
<td></td>
</tr>
<tr>
<td>w(Harry)</td>
<td></td>
</tr>
<tr>
<td>r(Harry)</td>
<td></td>
</tr>
<tr>
<td>w(Larry)</td>
<td></td>
</tr>
<tr>
<td>commit</td>
<td>commit</td>
</tr>
</tbody>
</table>

$1000 to Harry

$2000 to Larry
Anomalies: overwriting uncommitted data

T1_________________________T2
r(Larry)
w(Larry)
    r(Harry)
w(Harry)
r(Harry)
w(Harry)
r(Larry)
w(Larry) commit
commit

The problem is that T1 has overridden the result of T2, while T2 has not yet been committed.
Anomalies of interleaving

• Reading uncommitted data
• Unrepeatable reads
• Overriding uncommitted data

None of these would happen if we were executing transactions one after another: **serial schedules**
Summarizing the Terminology

• A **transaction** (model) is a *sequence* of *r* and *w* actions on database elements.

• A **schedule** is a *sequence* of reads/writes actions performed by a DBMS: to achieve interleaving and at the same time preserve consistency.

• **Serial Schedule** = All actions for each transaction are consecutive.
  
  \[ r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B); \ldots \]

• **Serializable Schedule**: A schedule whose “**effect**” is equivalent to that of some serial schedule.

We will introduce a **sufficient condition** for serializability.
Equivalent schedules and conflicts

- Two transactions **conflict** if they access the same data (X or Y) and *at least one of them is a write*.

  - \( r_i(X); r_j(Y) \equiv r_j(Y); r_i(X) \) (even when \( X=Y \))
    - No conflict

  - We can flip \( r_i(X); w_j(Y) \) as long as \( X \neq Y \)
    - No conflict

  - However, \( r_i(X); w_j(X) \neq w_j(X); r_i(X) \)
    - Conflict!

  - We can flip \( w_i(X); w_j(Y) \); provided \( X \neq Y \)
    - No conflict

  - However, \( w_i(X); w_j(X) \neq w_j(X); w_i(X) \);
    - Conflict!
    - The final value of \( X \) may be different depending on which write occurs last.
Conflicts: summary

Summarizing, there is a conflict if one of these two conditions hold.

1. A read and a write of the same X, or
2. Two writes of the same X

- Such actions conflict in general and may not be swapped in order.
- All other events (reads/writes) may be swapped without changing the effect of the schedule.

A schedule is conflict-serializable if it can be converted into a serial schedule by a series of non-conflicting swaps of adjacent elements.
Example:

What can we say about the original schedule?

\[
\begin{align*}
& r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B) \\
\end{align*}
\]

Non-conflicting swaps:

\[
\begin{align*}
& r_1(A); w_1(A); r_2(A); r_1(B); \underline{w_2(A)}; w_1(B); r_2(B); w_2(B) \\
& r_1(A); w_1(A); r_1(B); \underline{r_2(A)}; w_2(A); \underline{w_1(B)}; r_2(B); w_2(B) \\
& r_1(A); w_1(A); r_1(B); r_2(A); \underline{w_1(B)}; \underline{w_2(A)}; r_2(B); w_2(B) \\
& r_1(A); w_1(A); r_1(B); \underline{w_1(B)}; \underline{r_2(A)}; w_2(A); r_2(B); w_2(B) \\
\end{align*}
\]

Result: serial schedule
Conflict-serializability

**Sufficient** condition for serializability but **not necessary**.

**Example**

**S1**: \( w_1(Y); w_1(X); w_2(Y); w_2(X); w_3(X); \) -- This is serial

**S2**: \( w_1(Y); w_2(Y); w_2(X); w_1(X); w_3(X); \)

This is called **view-serializable**, and requires from scheduler to understand what each action is doing, not just its type.

S2 isn’t conflict-serializable, but it is serializable. It has the same effect as S1.

> Intuitively, the values of X written by T1 and T2 have no effect, since T3 overwrites them.
Serializability/precedence Graphs

- Non-swappable pairs of actions represent potential conflicts between transactions.
- The existence of non-swappable actions enforces an ordering on the transactions that include these actions.

We can represent this order by a graph:
- **Nodes**: transactions \( \{T_1, \ldots, T_k\} \)
- **Arcs**: There is a directed edge from \( T_i \) to \( T_j \) if they have conflicting access to the same database element \( X \) and \( T_i \) is first:
  
  written \( T_i <_s T_j \).
Precedence graphs: example 1

\[ r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B) \]

Note the following:

- \( w_1(B) <_S r_2(B) \)
- \( r_2(A) <_S w_3(A) \)

- These are conflicts since they contain a read/write on the same element
- They cannot be swapped.

Therefore \( T_1 < T_2 < T_3 \)
Precedence graphs: example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$

Note the following:
- $r_1(B) <_S w_2(B)$
- $w_2(A) <_S w_3(A)$
- $r_2(B) <_S w_1(B)$

Here, we have $T_1 < T_2 < T_3$, but we also have $T_2 < T_1$

Not conflict-serializable
Precedence graphs test for conflict-serializability

- **If there is a cycle in the graph**, then there is **no** serial schedule which is conflict-equivalent to S.
  - Each arc represents a requirement on the order of transactions in a conflict-equivalent *serial schedule*.
  - A cycle puts too many requirements on any *linear order* of transactions.

- **If there is no cycle in the graph**, then **any topological order** of the graph suggests a conflict-equivalent schedule.

---

*A topological ordering of a directed acyclic graph (DAG) is a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges,*
If a Precedence-Graph is acyclic, it represents a conflict-serializable schedule

**Proof idea:**
if the precedence graph is acyclic, then we can swap actions to form serial schedule.

- Given that the precedence graph is acyclic, there exists $T_k$ such that there is no $T_m$ that $T_k$ depends on, i.e. $T_k$ does not have any incoming edges.

- In this case, we can bring all actions of $T_k$ to the front of the schedule.
  
  (Actions of $T_k$)(Actions of the other $n-1$ transactions)

- The **tail** is a precedence graph that is the same as the original without $T_k$, i.e. it has $n-1$ nodes.

- Repeat for the tail.
Enforcing serializability by locks

- If scheduler allows multiple transactions access the same element, this may result in non-serializable schedule.
- To prevent this, before reading or writing an element $X$, a transaction $T_i$ requests a lock on $X$ from the scheduler.
- The scheduler can either grant the lock to $T_i$ or make $T_i$ wait for the lock.
- If granted, $T_i$ should eventually unlock (release) the lock on $X$.

Notations:

$L_i(X) = “\text{transaction } T_i \text{ requests a lock on } X”$

$u_i(X) \text{ (or } uL_i(X) \text{ )} = “T_i \text{ unlocks/releases the lock on } X”$
Legal schedule with locks

Schedule with locks - constraints:

Consistency of Transactions:

- Read or write X only when hold a lock on X.
  \[ r_i(X) \] or \[ w_i(X) \] must be preceded by some \[ L_i(X) \] with no intervening \[ u_i(X) \].
- If \( T_i \) locks X, \( T_i \) must eventually unlock X.
  Every \( L_i(X) \) must be followed by \( u_i(X) \).

Legality of Schedules:

- Two transactions may not have locked the same element X without one having first released the lock.
  A schedule with \( L_i(X) \) cannot have another \( L_j(X) \) until \( u_i(X) \) appears in between.
### Legal schedule doesn’t mean serializable!

- **T1 adds 100 to both A and B**
- **T2 doubles both A and B**
- **Consistency constraint: A=B, and should be 250 for both by the end**

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>25</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L₁(A); r₁(A)</th>
<th>A = A + 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>w₁(A); u₁(A)</td>
<td>125</td>
</tr>
</tbody>
</table>

T1 unlocks A so T2 is free to lock it

<table>
<thead>
<tr>
<th>L₂(A); r₂(A)</th>
<th>A = A * 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>w₂(A); u₂(A)</td>
<td>250</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L₂(B); r₂(B)</th>
<th>B = B * 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>w₂(B); u₂(B)</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L₁(B); r₁(B)</th>
<th>B = B + 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>w₁(B); u₁(B)</td>
<td>150</td>
</tr>
</tbody>
</table>
Two-Phase Locking

There is a simple condition, which guarantees conflict-serializability:

In every transaction, all lock requests (phase 1) precede all unlock requests (phase 2).

<table>
<thead>
<tr>
<th></th>
<th>T₁</th>
<th>T₂</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>L₁(A); r₁(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A = A + 100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w₁(A); L₁(B); u₁(A)</td>
<td>125</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>L₂(A); r₂(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A = A * 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w₂(A)</td>
<td>250</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>L₂(B) Denied</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r₁(B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B = B + 100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w₁(B); u₁(B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>L₂(B); u₂(A); r₂(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B = B * 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w₂(B); u₂(B)</td>
<td>250</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A legal schedule with 2PL locking is conflict-serializable

- **Proof idea:** in 2PL each transaction that starts unlocking, has already acquired all locks on its elements, and thus all its actions can be moved to the front of the schedule.

- **Proof by contradiction:** $S=\{T_1,\ldots,T_n\}$. Find the first transaction, say $T_1$, to perform an unlock action, say $u_1(X)$. We show that the r/w actions of $T_1$ can be moved to the front of the other transactions without conflict.

- Consider some action such as $w_1(Y)$. Let assume that it is preceded by some conflicting action $w_2(Y)$ or $r_2(Y)$. In such a case we cannot swap them.
  - If so, then $u_2(Y)$ and $L_1(Y)$ must interleave, as $w_2(Y)\ldots u_2(Y)\ldots L_1(Y)\ldots w_1(Y)$
  - Since $T_1$ is the first to unlock, $u_1(X)$ appears before $u_2(Y)$.
  - But then $L_1(Y)$ appears after $u_1(X)$, contradicting 2PL.
A legal schedule with 2PL locking is conflict-serializable

**Conclusion:** \(w_1(Y)\) can slide forward in the schedule without conflict; similar argument for a \(r_1(Y)\) action.
Simple locks are too restrictive

- While simple locks + 2PL guarantee conflict-serializability, they do not allow two readers of DB element X at the same time.

- But having multiple readers is not a problem for conflict-serializability (since read actions commute).
Solution: Two kinds of locks:

I. Shared lock $s_{L_i}(X)$ allows $T_i$ to read, but not write $X$. It prevents other transactions from writing $X$ but not from reading $X$.

II. Exclusive lock $x_{L_i}(X)$ allows $T_i$ to read and/or write $X$. No other transaction may read or write $X$. 
Shared/Exclusive Locks: changes

Consistency of transactions:
- A read $r_i(X)$ must be preceded by $sL_i(X)$ or $xL_i(X)$, with no intervening $u_i(X)$.
- A write $w_i(X)$ must be preceded by $xL_i(X)$, with no intervening $u_i(X)$.

Legal schedules:
- No two exclusive locks on the same element.
  - If $xL_i(X)$ appears in a schedule, then there cannot be a $xL_j(X)$ until after a $u_i(X)$ appears.
- No shared locks on exclusively locked element.
  - If $xL_i(X)$ appears, there can be no $sL_j(X)$ until after $u_i(X)$.
- No writing in shared lock mode
  - If $sL_i(X)$ appears, there can be no $w_j(X)$ until after $u_i(X)$.

2PL condition:
- No transaction may have a $sL(X)$ or $xL(X)$ after a $u(Y)$. 
Scheduler rules for shared/exclusive locks

- When there is more than one kind of lock, the scheduler needs a rule that says “if there is already a lock of type A on DB element X, can I grant a lock of type B on X?”
- The compatibility matrix answers the question.

Compatibility Matrix for Shared/Exclusive Locks

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>X</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Scheduling with locks: example

\r1(A); r2(B); r3(C); r1(B); r2(C); r3(D); w1(A); w2(B); w3(C);\n
<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>xl(A); r1(A)</td>
<td>xl(B); r2(B)</td>
<td>xl(C); r3(C)</td>
</tr>
<tr>
<td>sl(B)</td>
<td>denied</td>
<td>sl(C) denied</td>
<td>sl(D); r3(D); ul(D)</td>
</tr>
<tr>
<td></td>
<td>w1(A);</td>
<td>w2(B);</td>
<td>w3(C); ul(C)</td>
</tr>
<tr>
<td></td>
<td>sl(B); r1(B);</td>
<td>sl(C); r2(C);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ul(A); ul(B)</td>
<td>ul(B); ul(B)</td>
<td></td>
</tr>
</tbody>
</table>
Upgrading Locks

• Instead of taking an exclusive lock immediately, a transaction can take a shared lock on X, read X, and then upgrade the lock to exclusive so that it can write X.

\[
\begin{array}{l}
T_1 \\
\hline
sl_1(A); r_1(A); \\
sl_1(B); r_1(B); \\
xl_1(B) \text{ Denied} \\
xl_1(B); w_1(B); \\
u_1(A); u_2(B);
\end{array}
\quad
\begin{array}{l}
T_2 \\
\hline
sl_2(A); r_2(A); \\
sl_2(B); r_2(B); \\
\quad \\
u_2(A); u_2(B)
\end{array}
\]

Upgrading Locks allows more concurrent operations:

Had T1 asked for an exclusive lock on B before reading B, the request would have been denied, because T2 already has a shared lock on B.
Scheduling with upgrade locks: example

```
r1(A); r2(B); r3(C); r1(B); r2(C); r3(D); w1(A); w2(B); w3(C);
```

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>sl(A); r1(A);</td>
<td>sl(B); r2(B);</td>
<td>sl(C); r3(C);</td>
</tr>
<tr>
<td>sl(B); r1(B);</td>
<td>sl(C); r2(C);</td>
<td>sl(D); r3(D);</td>
</tr>
<tr>
<td>xl(A); w1(A); ul(A); ul(B);</td>
<td>xl(B); w2(B); ul(B); ul(C);</td>
<td>xl(C); w3(C); ul(C); ul(D);</td>
</tr>
</tbody>
</table>

Compared to slide 42: no waiting
Possibility of Deadlocks

**Example:** T1 and T2 each reads X and later writes X.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{sl}_1(X)$</td>
<td>$\text{sl}_2(X)$</td>
</tr>
<tr>
<td>$\text{x}_1(X)$ denied</td>
<td>$\text{x}_2(X)$ denied</td>
</tr>
</tbody>
</table>

**Problem:** when we allow upgrades, it is easy to get into a deadlock situation.

“When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.”
Solution: Update Locks

Update lock $ul_i(X)$
- Only an update lock (not shared lock) can be upgraded to exclusive lock (if there are no shared locks anymore).
- A transaction that will read and later on write some element $A$, asks initially for an update lock on $A$, and then asks for an exclusive lock on $A$. Such transaction doesn’t ask for a shared lock on $A$.

Legal schedules
- Read action permitted when there is either a shared or update lock.
- An update lock can be granted while there is a shared lock, but the scheduler will not grant a shared lock when there is an update lock.

2PL condition
- No transaction may have an $sl(X)$, $ul(X)$ or $xl(X)$ after a $u(Y)$.
# Update Locks: scheduler rules

## Compatibility Matrix for Shared/Exclusive/Update Locks

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>X</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>X</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>U</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Schedule with update locks: example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s!L(A); \ r(A)$</td>
<td>$u!L(A); \ r(A)$</td>
<td>$s!L(A) \ Denied$</td>
</tr>
<tr>
<td>$u(A)$</td>
<td>$x!L(A) \ Denied$</td>
<td></td>
</tr>
<tr>
<td>$x!L(A); \ w(A)$</td>
<td>$u(A)$</td>
<td>$s!L(A); \ r(A)$</td>
</tr>
<tr>
<td>$u(A)$</td>
<td></td>
<td>$u(A)$</td>
</tr>
</tbody>
</table>
(No) Deadlock Example

$T_1$ and $T_2$ each read $X$ and later write $X$.

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$sl_1(X)$</td>
<td>$sl_2(X)$</td>
</tr>
<tr>
<td>$xl_1(X)$ Denied</td>
<td>$xl_2(X)$ Denied</td>
</tr>
</tbody>
</table>

Deadlock when using $SL$ and $XL$ locks only.

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ul_1(X); r_1(X);$</td>
<td>$ul_2(X)$ Denied</td>
</tr>
<tr>
<td>$xl_1(X); w_1(X); u_1(X);$</td>
<td>$ul_2(X); r_2(X);$</td>
</tr>
<tr>
<td>$xl_2(X); w_2(X); u_2(X);$</td>
<td>$xl_2(X); w_2(X); u_2(X);$</td>
</tr>
</tbody>
</table>

Fine when using update locks.
Scheduling with 3 types of locks: example

\[ r_1(A); r_2(B); r_3(C); r_1(B); r_2(C); r_3(D); w_1(A); w_2(B); w_3(C); \]

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>uL(A); r1(A);</td>
<td>uL(B); r2(B);</td>
<td>uL(C); r3(C);</td>
</tr>
<tr>
<td>sL(B); <strong>denied</strong></td>
<td>sL(C); <strong>denied</strong></td>
<td>sL(D); r3(D);</td>
</tr>
<tr>
<td>xl(A); w1(A);</td>
<td>xl(B); w2(B);</td>
<td>xl(C); w3(C);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>uL(D); uL(C);</td>
</tr>
<tr>
<td>sL(B); r1(B);</td>
<td>sL(C); r2(C);</td>
<td></td>
</tr>
<tr>
<td>uL(A); uL(B);</td>
<td>uL(B); uL(C);</td>
<td></td>
</tr>
</tbody>
</table>
Benefits of Update Locks

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
<th>T7</th>
<th>T8</th>
<th>T9</th>
</tr>
</thead>
<tbody>
<tr>
<td>sl(A);r(A)</td>
<td>sl(A);r(A)</td>
<td>sl(A);r(A)</td>
<td>sl(A);r(A)</td>
<td>ul(A);r(A)</td>
<td>sl(A);r(A)</td>
<td>sl(A);r(A)</td>
<td>sl(A);r(A)</td>
<td>sl(A);r(A)</td>
</tr>
</tbody>
</table>

sl – shared lock
ul – update lock
xl – exclusive lock
u - unlock
Locking granularity

• Should we lock the whole table or just a single row? What’s a database object? What should be the locking granularity?

```sql
SELECT min(year)
FROM Movies;
```

• What happens if we insert a new tuple with the smallest year?

• **Phantom problem**: A transaction retrieves a collection of tuples twice and sees new tuples (phantoms)

• Two-phase locking of the entire table prevents phantoms, but **table-level** locking can be restrictive in cases where transaction accesses only a few tuples via an index.
Transaction control in SQL

Gives control over the locking overhead

• **Access mode:**
  - READ ONLY
  - READ WRITE

• **Isolation level** (to which extent transaction is exposed to actions of other transactions):
  - SERIALIZABLE (Default)
  - REPEATABLE READ
  - READ COMMITED
  - READ UNCOMMITTED
Transaction Isolation Levels

SET TRANSACTION ISOLATION LEVEL X READ WRITE

Where X can be
- SERIALIZABLE (Default)
- REPEATABLE READ
- READ COMMITED
- READ UNCOMMITED

Decreasing isolation level

With a scheduler based on locks:
- A **SERIALIZABLE** transaction obtains locks before reading and writing objects, including locks on sets (e.g. table) of objects that it requires to be unchangeable and holds them until the end, according to 2PL.

- A **REPEATABLE READ** transaction sets the same locks as a **SERIALIZABLE** transaction, except that it doesn’t lock sets of objects, but only individual objects.
Transaction Isolation Levels

- A **READ COMMITTED** transaction T obtains exclusive locks before writing objects and keeps them until the end. However, it obtains shared locks before reading values and then immediately releases them; That is to ensure that the transaction that last modified the values is complete.
  - T reads only the changes made by committed transactions.
  - No value written by T is changed by any other transaction until T is completed.
  - However, a value read by T may well be modified by another transaction (which eventually commits) while T is still in progress.
  - T is also exposed to the *phantom* problem.

- A **READ UNCOMMITTED** transaction doesn’t obtain any lock at all. So, it can read data that is being modified. Such transactions are allowed to be READ ONLY only.
Summary: ACID transactions

- **Consistency**: Database constraints preserved. Transaction, executed completely, takes database from one consistent state to another: *serializable schedules*
- **Isolation**: It appears to the user as if only one process executes at a time: *locking*

Next we talk how to ensure:
- **Atomicity**: Whole transaction or none is done.
- **Durability**: Effects of a process survive a crash.