CSE 544: Principles of Database Systems

Lecture 18: Concurrency Control
Project Presentations

Friday, June 7th, 9:30-2:30, in CSE 405 (details TBA)

What to include:
• Describe the problem:
  – why is it important, why is it non-trivial
• Overview prior approaches,
  – related work
• Your approach
• Your results
  – theoretical, empirical, experimental
• Discuss their significance
  – do they work? do they solve the problem you set out to do? do they improve over existing work?
• Conclusions

Rule of thumb: 1 slide / minute, less slack. 15’ ➔ 12 slides.
Reading Material

Main textbook (Ramakrishnan and Gehrke):
• Chapters 16, 17, 18

More background material: Garcia-Molina, Ullman, Widom:
• Chapters 17.2, 17.3, 17.4
• Chapters 18.1, 18.2, 18.3, 18.8, 18.9
Concurrent Control

- Multiple concurrent transactions $T_1$, $T_2$, …
- They read/write common elements $A_1$, $A_2$, …
- How can we prevent unwanted interference?

The SCHEDULER is responsible for that
Schedules

A schedule is a sequence of interleaved actions from all transactions
Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A, s)</td>
</tr>
<tr>
<td>( t := t + 100 )</td>
<td>( s := s \times 2 )</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A, s)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>READ(B, s)</td>
</tr>
<tr>
<td>( t := t + 100 )</td>
<td>( s := s \times 2 )</td>
</tr>
<tr>
<td>WRITE(B, t)</td>
<td>WRITE(B, s)</td>
</tr>
</tbody>
</table>
## A Serial Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A, s)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>READ(B, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B, t)</td>
<td>WRITE(B, s)</td>
</tr>
</tbody>
</table>
Serializable Schedule

A schedule is *serializable* if it is equivalent to a serial schedule.
A Serializable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A,s)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>READ(B,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>WRITE(B,s)</td>
</tr>
</tbody>
</table>

This is NOT a serial schedule, but is *serializable*
A Non-Serializable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A,s)</td>
</tr>
<tr>
<td></td>
<td>READ(B,s)</td>
</tr>
<tr>
<td></td>
<td>s := s*2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B,s)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td></td>
</tr>
<tr>
<td>t := t+100</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td></td>
</tr>
</tbody>
</table>
Serializable Schedules

- The role of the scheduler is to ensure that the schedule is serializable

**Q:** Why not run only serial schedules? I.e. run one transaction after the other?
Serializable Schedules

• The role of the scheduler is to ensure that the schedule is serializable

Q: Why not run only serial schedules? I.e. run one transaction after the other?

A: Because of very poor throughput due to disk latency.

Lesson: main memory databases may do serial schedules only
A Serializable Schedule

We don’t expect the scheduler to schedule this

Schedule is serializable because \( t = t + 100 \) and \( s = s + 200 \) commute

\[
\begin{align*}
\text{T1} & \quad \text{T2} \\
\text{READ}(A, t) & \quad \text{READ}(A, s) \\
t := t + 100 & \quad s := s + 200 \\
\text{WRITE}(A, t) & \quad \text{WRITE}(A, s) \\
\text{READ}(B, t) & \quad \text{READ}(B, s) \\
t := t + 100 & \quad s := s + 200 \\
\text{WRITE}(B, t) & \quad \text{WRITE}(B, s)
\end{align*}
\]
Ignoring Details

• Assume worst case updates:
  – We never commute actions done by transactions

• As a consequence, we only care about reads and writes
  – Transaction = sequence of R(A)’s and W(A)’s

\[
T_1: r_1(A); w_1(A); r_1(B); w_1(B) \\
T_2: r_2(A); w_2(A); r_2(B); w_2(B)
\]
Conflicts

- Write-Read – WR
- Read-Write – RW
- Write-Write – WW
Conflicts

Two actions by same transaction $T_i$:
\[ r_i(X); w_i(Y) \]

Two writes by $T_i$, $T_j$ to same element
\[ w_i(X); w_j(X) \]

Read/write by $T_i$, $T_j$ to same element
\[ w_i(X); r_j(X) \]
\[ r_i(X); w_j(X) \]

A “conflict” means: you can’t swap the two operations
Conflict Serializability

- A schedule is *conflict serializable* if it can be transformed into a serial schedule by a series of swappings of adjacent non-conflicting actions.

Example:

\[
\begin{align*}
&\text{r}_1(\text{A}); \text{w}_1(\text{A}); \text{r}_2(\text{A}); \text{w}_2(\text{A}); \text{r}_1(\text{B}); \text{w}_1(\text{B}); \text{r}_2(\text{B}); \text{w}_2(\text{B}) \\
\downarrow \\
&\text{r}_1(\text{A}); \text{w}_1(\text{A}); \text{r}_1(\text{B}); \text{w}_1(\text{B}); \text{r}_2(\text{A}); \text{w}_2(\text{A}); \text{r}_2(\text{B}); \text{w}_2(\text{B})
\end{align*}
\]
Testing for Conflict-Serializability

Precedence graph:

• A node for each transaction $T_i$,
• An edge from $T_i$ to $T_j$ whenever an action in $T_i$ conflicts with, and comes before an action in $T_j$

• The schedule is serializable iff the precedence graph is acyclic
Example 1

\[ r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B) \]
Example 1

This schedule is conflict-serializable
Example 2

\[ r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B) \]
Example 2

This schedule is NOT conflict-serializable

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

• A serializable schedule need not be conflict serializable, even under the “worst case update” assumption

\[ w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y); \]

Is this schedule conflict-serializable?
View Equivalence

• A serializable schedule need not be conflict serializable, even under the “worst case update” assumption

\[ w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y); \]

Is this schedule conflict-serializable? No...
View Equivalence

- A serializable schedule need not be conflict serializable, even under the “worst case update” assumption

Lost write

Equivalent, but not conflict-equivalent
View Equivalence

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1(X)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>W2(X)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W2(Y)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CO2</td>
<td></td>
</tr>
<tr>
<td>W1(Y)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CO1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1(X)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>W1(Y)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CO1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2(X)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W2(Y)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CO2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W3(Y)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CO3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Lost

Serializable, but not conflict serializable
Two schedules $S$, $S'$ are view equivalent if:

- If $T$ reads an initial value of $A$ in $S$, then $T$ reads the initial value of $A$ in $S'$.

- If $T$ reads a value of $A$ written by $T'$ in $S$, then $T$ reads a value of $A$ written by $T'$ in $S'$.

- If $T$ writes the final value of $A$ in $S$, then $T$ writes the final value of $A$ in $S'$. 
View-Serializability

A schedule is *view serializable* if it is view equivalent to a serial schedule

Remark:

• If a schedule is *conflict serializable*, then it is also *view serializable*
• But not vice versa
Schedules with Aborted Transactions

• When a transaction aborts, the recovery manager undoes its updates

• But some of its updates may have affected other transactions!
Schedules with Aborted Transactions

<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>W(B)</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

Abort

Cannot abort T1 because cannot undo T2
A schedule is *recoverable* if:

- It is conflict-serializable, and
- Whenever a transaction $T$ commits, all transactions who have written elements read by $T$ have already committed
Recoverable Schedules

Nonrecoverable

Recoverable
## Recoverable Schedules

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td>R(A)</td>
<td>W(A)</td>
<td>R(B)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
<td>R(B)</td>
<td>W(B)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R(B)</td>
<td>W(B)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R(C)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R(C)</td>
<td>W(C)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R(D)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R(D)</td>
<td>W(D)</td>
</tr>
</tbody>
</table>

Abort

**How do we recover?**
Cascading Aborts

• If a transaction $T$ aborts, then we need to abort any other transaction $T'$ that has read an element written by $T$.

• A schedule avoids cascading aborts if whenever a transaction reads an element, the transaction that has last written it has already committed.
## Avoiding Cascading Aborts

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(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>...</td>
</tr>
</tbody>
</table>

**With cascading aborts**

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

**Without cascading aborts**
Review of Schedules

Serializability

• Serial
• Serializable
• Conflict serializable
• View serializable

Recoverability

• Recoverable
• Avoids cascading deletes
Scheduler

• The scheduler:
  • Module that schedules the transaction’s actions, ensuring serializability

• Two main approaches
  • Pessimistic: locks
  • Optimistic: time stamps, MV, validation
Pessimistic Scheduler

Simple idea:

• Each element has a unique lock
• Each transaction must first acquire the lock before reading/writing that element
• If the lock is taken by another transaction, then wait
• The transaction must release the lock(s)
Notation

$l_i(A)$ = transaction $T_i$ acquires lock for element $A$

$u_i(A)$ = transaction $T_i$ releases lock for element $A$
A Non-Serializable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A,s)</td>
</tr>
<tr>
<td>READ(B,s)</td>
<td>READ(B,s)</td>
</tr>
<tr>
<td>s := s*2</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B,s)</td>
<td>WRITE(B,s)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>READ(B, t)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>t := t+100</td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>WRITE(B,t)</td>
</tr>
</tbody>
</table>
## Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L₁(A); READ(A, t)</td>
<td>L₂(A); READ(A, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t); U₁(A); L₁(B)</td>
<td>WRITE(A, s); U₂(A); L₂(B); DENIED…</td>
</tr>
</tbody>
</table>

READ(B, t)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B,t); U₁(B);</td>
<td>WRITE(B,s); U₂(B);</td>
</tr>
</tbody>
</table>

…GRANTED; READ(B,s)

Scheduler has ensured a conflict-serializable schedule
But…

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_1(A)$; READ(A, t)</td>
<td></td>
</tr>
<tr>
<td>$t := t+100$</td>
<td>$L_2(A)$; READ(A, s)</td>
</tr>
<tr>
<td>WRITE(A, t); $U_1(A)$</td>
<td>$s := s*2$</td>
</tr>
<tr>
<td></td>
<td>WRITE(A, s); $U_2(A)$</td>
</tr>
<tr>
<td>$L_1(B)$; READ(B, t)</td>
<td>$L_2(B)$; READ(B, s)</td>
</tr>
<tr>
<td>$t := t+100$</td>
<td>$s := s*2$</td>
</tr>
<tr>
<td>WRITE(B, t); $U_1(B)$</td>
<td>WRITE(B, s); $U_2(B)$</td>
</tr>
</tbody>
</table>

Locks did not enforce conflict-serializability !!! What’s wrong ?
Two Phase Locking (2PL)

The 2PL rule:

• In every transaction, all lock requests must precede all unlock requests

• This ensures conflict serializability! (will prove this shortly)
Example: 2PL transactions

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L₁(A); L₁(B); READ(A, t)</td>
<td>L₂(A); READ(A,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t); U₁(A)</td>
<td>WRITE(A,s);</td>
</tr>
<tr>
<td></td>
<td>L₂(B); DENIED…</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>…GRANTED; READ(B,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B,t); U₁(B);</td>
<td>WRITE(B,s); U₂(A); U₂(B);</td>
</tr>
</tbody>
</table>

Now it is conflict-serializable
Two Phase Locking (2PL)

**Theorem:** 2PL ensures conflict serializability
Two Phase Locking (2PL)

**Theorem:** 2PL ensures conflict serializability

**Proof.** Suppose not: then there exists a cycle in the precedence graph.

Then there is the following **temporal** cycle in the schedule:

- $U_1(A) \rightarrow L_2(A)$
- $L_2(A) \rightarrow U_2(B)$
- $U_2(B) \rightarrow L_3(B)$
- $L_3(B) \rightarrow U_3(C)$
- $U_3(C) \rightarrow L_1(C)$
- $L_1(C) \rightarrow U_1(A)$

Contradiction
A New Problem: Non-recoverable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_1(A); L_1(B)$; READ(A, t)</td>
<td>$L_2(A); READ(A,s)$</td>
</tr>
<tr>
<td>$t := t+100$</td>
<td>$s := s*2$</td>
</tr>
<tr>
<td>WRITE(A, t); $U_1(A)$</td>
<td>WRITE(A,s);</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>$L_2(B); DENIED…$</td>
</tr>
<tr>
<td>$t := t+100$</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t); $U_1(B)$</td>
<td>…GRANTED; READ(B,s)</td>
</tr>
<tr>
<td></td>
<td>$s := s*2$</td>
</tr>
<tr>
<td></td>
<td>WRITE(B,s); $U_2(A); U_2(B);$</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

Abort
What about Aborts?

• 2PL enforces conflict-serializable schedules

• But does not enforce recoverable schedules
Strict 2PL

• Strict 2PL: All locks held by a transaction are released when the transaction is completed.

• Schedule is recoverable
  – Transactions commit only after all transactions whose changes they read also commit.

• Schedule avoids cascading aborts
  – Transactions read only after the txn that wrote that element committed.

• Schedule is strict: read book.
Lock Modes

Standard:
• $S$ = shared lock (for READ)
• $X$ = exclusive lock (for WRITE)

Lots of fancy locks:
• $U$ = update lock
  – Initially like $S$
  – Later may be upgraded to $X$
• $I$ = increment lock (for $A := A + \text{something}$)
  – Increment operations commute
Lock Granularity

- **Fine granularity locking** (e.g., tuples)
  - High concurrency
  - High overhead in managing locks

- **Coarse grain locking** (e.g., tables, predicate locks)
  - Many false conflicts
  - Less overhead in managing locks

- **Alternative techniques**
  - Hierarchical locking (and intentional locks) [commercial DBMSs]
  - Lock escalation
Deadlocks

• Trasaction $T_1$ waits for a lock held by $T_2$;
• But $T_2$ waits for a lock held by $T_3$;
• While $T_3$ waits for . . . .
• . . .
• . . .and $T_{73}$ waits for a lock held by $T_1$ !!
Deadlocks

• When T1 waits for T2, which waits for T3, which waits for T4, …, which waits for T1 – cycle!

• Deadlock avoidance
  – Acquire locks in pre-defined order
  – Acquire all locks at once before starting

• Deadlock detection
  – Timeouts
  – Wait-for graph (this is what commercial systems use)
The Locking Scheduler

Task 1:
- Add lock/unlock requests to transactions
  - Examine all READ(A) or WRITE(A) actions
  - Add appropriate lock requests
  - Ensure Strict 2PL!
The Locking Scheduler

Task 2:

Execute the locks accordingly

• Lock table: a big, critical data structure in a DBMS!
• When a lock is requested, check the lock table
  – Grant, or add the transaction to the element’s wait list
• When a lock is released, re-activate a transaction from its wait list
• When a transaction aborts, release all its locks
• Check for deadlocks occasionally
Lock Performance

Throughput vs. # Active Transactions

thrashing

Why?
The Tree Protocol

• An alternative to 2PL, for tree structures
• E.g. B-trees (the indexes of choice in databases)

• Because
  – Indexes are hot spots!
  – 2PL would lead to great lock contention
The Tree Protocol

Rules:
• The first lock may be any node of the tree
• Subsequently, a lock on a node A may only be acquired if the transaction holds a lock on its parent B
• Nodes can be unlocked in any order (no 2PL necessary)
• “Crabbing”
  – First lock parent then lock child
  – Keep parent locked only if may need to update it
  – Release lock on parent if child is not full

• The tree protocol is NOT 2PL, yet ensures conflict-serializability!
Phantom Problem

• So far we have assumed the database to be a static collection of elements (=tuples)

• If tuples are inserted/deleted then the phantom problem appears
## Phantom Problem

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
</table>
| SELECT *  
FROM Product  
WHERE color='blue' | INSERT INTO Product(name, color)  
VALUES ('gizmo','blue') |
| SELECT *  
FROM Product  
WHERE color='blue' |
Phantom Problem

Suppose there are two blue products, X1, X2:

R1(X1), R1(X2), W2(X3), R1(X1), R1(X2), R1(X3)

T1

| SELECT *  
| FROM Product 
| WHERE color='blue' |

T2

| INSERT INTO Product(name, color) VALUES ('gizmo', 'blue') |

| SELECT *  
| FROM Product 
| WHERE color='blue' |
Phantom Problem

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT *</td>
<td>INSERT INTO Product(name, color) VALUES ('gizmo','blue')</td>
</tr>
<tr>
<td>FROM Product</td>
<td></td>
</tr>
<tr>
<td>WHERE color='blue'</td>
<td></td>
</tr>
</tbody>
</table>

Suppose there are two blue products, X1, X2:

R1(X1), R1(X2), W2(X3), R1(X1), R1(X2), R1(X3)

This is conflict serializable! What’s wrong??
Suppose there are two blue products, X1, X2:

Not serializable due to **phantoms**
Phantom Problem

• A “phantom” is a tuple that is invisible during part of a transaction execution but not invisible during the entire execution

• In our example:
  – T1: reads list of products
  – T2: inserts a new product
  – T1: re-reads: a new product appears!
Phantom Problem

• In a **static** database:
  – Conflict serializability implies serializability

• In a **dynamic** database, this may fail due to phantoms

• Strict 2PL guarantees conflict serializability, but not serializability
Dealing With Phantoms

• Lock the entire table, or
• Lock the index entry for ‘blue’
  – If index is available
• Or use predicate locks
  – A lock on an arbitrary predicate

Dealing with phantoms is expensive!
Isolation Levels in SQL

1. “Dirty reads”
   SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

2. “Committed reads”
   SET TRANSACTION ISOLATION LEVEL READ COMMITTED

3. “Repeatable reads”
   SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

4. Serializable transactions
   SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

ACID
1. Isolation Level: Dirty Reads

- “Long duration” WRITE locks
  - Strict 2PL
- No READ locks
  - Read-only transactions are never delayed

Possible pbs: dirty and inconsistent reads
2. Isolation Level: Read Committed

- “Long duration” WRITE locks
  - Strict 2PL

- “Short duration” READ locks
  - Only acquire lock while reading (not 2PL)

Unrepeatable reads
When reading same element twice, may get two different values
3. Isolation Level: Repeatable Read

• “Long duration” WRITE locks
  – Strict 2PL

• “Long duration” READ locks
  – Strict 2PL

This is not serializable yet !!!
4. Isolation Level Serializable

- “Long duration” WRITE locks
  - Strict 2PL
- “Long duration” READ locks
  - Strict 2PL
- Deals with phantoms too