ADMINISTRIVIA

• OQ6 Out
  • 6 questions
  • Due next Wednesday, 11:00pm

• HW7 Shortened
  • Parts 1 and 2 -- other material candidates for short answer, go over in section

• Course evaluations
  • https://uw.iasystem.org/survey/188771
  • As of before class -- 13%
ADMINISTRISTRIVIA

• HW8
  • Due Friday
  • Up to 3 late days on the submission
  • No benefit for keeping late days
CLASS OVERVIEW

Unit 1: Intro
Unit 2: Relational Data Models and Query Languages
Unit 3: Non-relational data
Unit 4: RDMBS internals and query optimization
Unit 5: Parallel query processing
Unit 6: DBMS usability, conceptual design

Unit 7: Transactions
  • Locking and schedules
  • Writing DB applications
TRANSACTIONS

We use database transactions everyday

• Bank $$$ transfers
• Online shopping
• Signing up for classes

For this class, a transaction is a series of DB queries

• Read / Write / Update / Delete / Insert
• Unit of work issued by a user that is independent from others
CHALLENGES

Want to execute many apps concurrently
  • All these apps read and write data to the same DB

Simple solution: only serve one app at a time
  • What’s the problem?

Want: multiple operations to be executed *atomically* over the same DBMS
WHAT CAN GO WRONG?

Manager: balance budgets among projects

- Remove $10k from project A
- Add $7k to project B
- Add $3k to project C

CEO: check company’s total balance

- SELECT SUM(money) FROM budget;

This is called a dirty / inconsistent read aka a WRITE-READ conflict
WHAT CAN GO WRONG?

App 1:
SELECT inventory FROM products WHERE pid = 1

App 2:
UPDATE products SET inventory = 0 WHERE pid = 1

App 1:
SELECT inventory * price FROM products WHERE pid = 1

This is known as an unrepeatable read aka READ-WRITE conflict
WHAT CAN GO WRONG?

Account 1 = $100
Account 2 = $100
Total = $200

• App 1:
  – Set Account 1 = $200
  – Set Account 2 = $0

• App 2:
  – Set Account 2 = $200
  – Set Account 1 = $0

• At the end:
  – Total = $200

• App 1: Set Account 1 = $200

• App 2: Set Account 2 = $200

• App 1: Set Account 2 = $0

• App 2: Set Account 1 = $0

• At the end:
  – Total = $0

This is called the lost update aka WRITE-WRITE conflict
WHAT CAN GO WRONG?

Paying for Tuition (Underwater Basket Weaving)

• Fill up form with your mailing address
• Put in debit card number (because you don’t trust the gov’t)
• Click submit
• Screen shows money deducted from your account
• [Your browser crashes]

Lesson:
Changes to the database should be ALL or NOTHING
TRANSACTIONS

Collection of statements that are executed atomically (logically speaking)

BEGIN TRANSACTION
    [SQL statements]
COMMIT or ROLLBACK (=ABORT)

[single SQL statement]

If BEGIN... missing, then TXN consists of a single instruction
KNOW YOUR TRANSACTIONS: ACID

Atomic
- State shows either all the effects of txn, or none of them

Consistent
- Txn moves from a DBMS state where integrity holds, to another where integrity holds
  - remember integrity constraints?

Isolated
- Effect of txns is the same as txns running one after another (i.e., looks like batch mode)

Durable
- Once a txn has committed, its effects remain in the database
**ATOMIC**
Definition: A transaction is ATOMIC if all its updates must happen or not at all.

Example: move $100 from A to B

- UPDATE accounts SET bal = bal - 100 WHERE acct = A;
- UPDATE accounts SET bal = bal + 100 WHERE acct = B;

- BEGIN TRANSACTION;
  UPDATE accounts SET bal = bal - 100 WHERE acct = A;
  UPDATE accounts SET bal = bal + 100 WHERE acct = B;
  COMMIT;
**ISOLATED**

- **Definition:**
  - An execution ensures that transactions are isolated, if the effect of each transaction is as if it were the only transaction running on the system.
**CONSISTENT**

Recall: integrity constraints govern how values in tables are related to each other

- Can be enforced by the DBMS, or ensured by the app

How consistency is achieved by the app:

- App programmer ensures that txns only takes a consistent DB state to another consistent state
- DB makes sure that txns are executed atomically

Can defer checking the validity of constraints until the end of a transaction
A transaction is durable if its effects continue to exist after the transaction and even after the program has terminated.

How?

- By writing to disk!
- More in 444
ROLLBACK TRANSACTIONS

If the app gets to a state where it cannot complete the transaction successfully, execute ROLLBACK

The DB returns to the state prior to the transaction

What are examples of such program states?
Again: by default each statement is its own txn

- Unless auto-commit is off then each statement starts a new txn
A schedule is a sequence of interleaved actions from all transactions
A **serial schedule** is one in which transactions are executed one after the other, in some sequential order.

Fact: nothing can go wrong if the system executes transactions serially.

- (up to what we have learned so far)
- But DBMS don’t do that because we want better overall system performance.
<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>

A and B are elements in the database. t and s are variables in txn source code.
## EXAMPLE OF A (SERIAL) SCHEDULE

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

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

**Time**

- READ(A, t)
- t := t+100
- WRITE(A, t)
- READ(B, t)
- t := t+100
- WRITE(B, t)
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>
</tbody>
</table>

| READ(B, t)                              | READ(B,s)                               |
| t := t+100                              | s := s*2                                |
| WRITE(B,t)                              | WRITE(B,s)                              |

This is a **serializable** schedule.
This is NOT a serial schedule
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>WRITE(B,s)</td>
<td>WRITE(B,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,t)</td>
</tr>
</tbody>
</table>
HOW DO WE KNOW IF A SCHEDULE IS SERIALIZABLE?

Notation:

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

Key Idea: Focus on conflicting operations
CONFLICTS

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

Read-Read?
CONFLICT SERIALIZABILITY

Conflicts: (i.e., swapping will change program behavior)

Two actions by same transaction $T_i$:

\[ r_i(X); w_i(Y) \]

Two writes by $T_i, T_j$ to same element:

\[ w_i(X); w_j(X) \]
\[ w_i(X); r_j(X) \]
\[ r_i(X); w_j(X) \]

Read/write by $T_i, T_j$ to same element
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.

Every conflict-serializable schedule is serializable.

The converse is not true (why?)
CONFLICT SERIALIZABILITY

Example:

\[ r_1(A); \ w_1(A); \ r_2(A); \ w_2(A); \ r_1(B); \ w_1(B); \ r_2(B); \ w_2(B) \]
CONFLICT
SERIALIZABILITY

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$
CONFLICT SERIALIZABILITY

Example:

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

\[ r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B) \]
CONFLICT SERIALIZABILITY

Example:

- $r_1(A); w_1(A);$ $r_2(A); w_2(A);$ $r_1(B); w_1(B);$ $r_2(B); w_2(B)$
- $r_1(A); w_1(A);$ $r_2(A);$ $r_1(B); w_2(A);$ $w_1(B);$ $r_2(B); w_2(B)$
- $r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$
Example:

\[
\begin{align*}
&\text{r}_1(A); w_1(A); r_2(A); r_1(B); w_1(B); r_2(B); w_2(B) \\
&\text{r}_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B) \\
&\text{r}_1(A); w_1(A); r_1(B); r_2(A); w_2(A); w_1(B); r_2(B); w_2(B) \\
&\text{r}_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)
\end{align*}
\]
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 conflict-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
**SCHEDULER**

*Scheduler* = the module that schedules the transaction’s actions, ensuring serializability

Also called *Concurrency Control Manager*

We discuss next how a scheduler may be implemented
IMPLEMENTING A SCHEDULER

Major differences between database vendors

Locking Scheduler

• Aka “pessimistic concurrency control”
• SQLite, SQL Server, DB2

Multiversion Concurrency Control (MVCC)

• Aka “optimistic concurrency control”
• Postgres, Oracle: Snapshot Isolation (SI)

We discuss only locking schedulers in this class
LOCKING 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)

By using locks scheduler ensures conflict-serializability
WHAT DATA ELEMENTS ARE LOCKED?

Major differences between vendors:

Lock on the entire database
  • SQLite

Lock on individual records
  • SQL Server, DB2, etc
CASE STUDY: SQLITE

SQLite is very simple

More info: http://www.sqlite.org/atomiccommit.html

Lock types

- READ LOCK (to read)
- RESERVED LOCK (to write)
- PENDING LOCK (wants to commit)
- EXCLUSIVE LOCK (to commit)
SQLite

Step 1: when a transaction begins

Acquire a **READ LOCK** (aka "SHARED" lock)

All these transactions may read happily

They all read data from the database file

If the transaction commits without writing anything, then it simply releases the lock
SQLite

Step 2: when one transaction wants to write

Acquire a RESERVED LOCK

May coexists with many READ LOCKs

Writer TXN may write; these updates are only in main memory; others don't see the updates

Reader TXN continue to read from the file

New readers accepted

No other TXN is allowed a RESERVED LOCK
Step 3: when writer transaction wants to commit, it needs *exclusive lock*, which can’t coexists with *read locks*

Acquire a **PENDING LOCK**

May coexists with old READ LOCKs

No new READ LOCKS are accepted

Wait for all read locks to be released

Why not write to disk right now?
Step 4: when all read locks have been released
Acquire the **EXCLUSIVE LOCK**
Nobody can touch the database now
All updates are written permanently to the database file

Release the lock and **COMMIT**
begin transaction

None → READ LOCK

first write

READ LOCK → RESERVED LOCK

commit requested

RESERVED LOCK → PENDING LOCK

no more read locks

PENDING LOCK → EXCLUSIVE LOCK

commit

commit executed
SCHEDULE ANOMALIES

What could go wrong if we didn’t have concurrency control:

• Dirty reads (including inconsistent reads)
• Unrepeatable reads
• Lost updates

Many other things can go wrong too
DIRTY READS

Write-Read Conflict

$T_1$: WRITE(A)

$T_1$: ABORT

$T_2$: READ(A)
INCONSISTENT READ

Write-Read Conflict

\[ T_1: \ A := 20; \ B := 20; \]
\[ T_1: \ \text{WRITE}(A) \]
\[ T_1: \ \text{WRITE}(B) \]

\[ T_2: \ \text{READ}(A) ; \]
\[ T_2: \ \text{READ}(B) ; \]
UNREPEATABLE READ

Read-Write Conflict

$T_1$: WRITE(A)

$T_2$: READ(A);

$T_2$: READ(A);
LOST UPDATE

Write-Write Conflict

$T_1$: READ(A)

$T_1$: A := A+5

$T_1$: WRITE(A)

$T_2$: READ(A);  

$T_2$: A := A*1.3  

$T_2$: WRITE(A);
MORE NOTATIONS

$L_i(A)$ = 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)</td>
<td>READ(A)</td>
</tr>
<tr>
<td>A := A + 100</td>
<td>A := A * 2</td>
</tr>
<tr>
<td>WRITE(A)</td>
<td>WRITE(A)</td>
</tr>
<tr>
<td></td>
<td>READ(B)</td>
</tr>
<tr>
<td></td>
<td>B := B * 2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B)</td>
</tr>
<tr>
<td></td>
<td>READ(B)</td>
</tr>
<tr>
<td></td>
<td>B := B + 100</td>
</tr>
<tr>
<td></td>
<td>WRITE(B)</td>
</tr>
</tbody>
</table>
EXAMPLE

T1

\[ L_1(A); \text{READ}(A) \]
\[ A := A + 100 \]
\[ \text{WRITE}(A); U_1(A); L_1(B) \]

T2

\[ L_2(A); \text{READ}(A) \]
\[ A := A \times 2 \]
\[ \text{WRITE}(A); U_2(A); L_2(B); \text{BLOCKED…} \]

READ(B)
\[ B := B + 100 \]
\[ \text{WRITE}(B); U_1(B); \]

…GRANTED; READ(B)
\[ B := B \times 2 \]
\[ \text{WRITE}(B); U_2(B); \]

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); \text{READ}(A) )</td>
<td>( L_2(A); \text{READ}(A) )</td>
</tr>
<tr>
<td>( A := A+100 )</td>
<td>( A := A\times 2 )</td>
</tr>
<tr>
<td>( \text{WRITE}(A); U_1(A) )</td>
<td>( \text{WRITE}(A); U_2(A) )</td>
</tr>
<tr>
<td>( L_1(B); \text{READ}(B) )</td>
<td>( L_2(B); \text{READ}(B) )</td>
</tr>
<tr>
<td>( B := B+100 )</td>
<td>( B := B\times 2 )</td>
</tr>
<tr>
<td>( \text{WRITE}(B); U_1(B) )</td>
<td>( \text{WRITE}(B); U_2(B) )</td>
</tr>
</tbody>
</table>

Locks did not enforce conflict-serializability !!! What’s wrong?
The 2PL rule:

In every transaction, all lock requests must precede all unlock requests.
### EXAMPLE: 2PL TRANSACTIONS

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L_1(A); L_1(B); ) READ(A)</td>
<td>L(_2)(A); READ(A)</td>
</tr>
<tr>
<td>A := A + 100</td>
<td>A := A*2</td>
</tr>
<tr>
<td>WRITE(A); U(_1)(A)</td>
<td>WRITE(A); U(_1)(A)</td>
</tr>
<tr>
<td>READ(B)</td>
<td>L(_2)(B); BLOKKED…</td>
</tr>
<tr>
<td>B := B + 100</td>
<td>…GRANTED; READ(B)</td>
</tr>
<tr>
<td>WRITE(B); U(_1)(B)</td>
<td>B := B*2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B); U(_2)(A); U(_2)(B);</td>
</tr>
</tbody>
</table>

Now it is conflict-serializable
Theorem: 2PL ensures conflict serializability
Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.
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:
**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: \( U_1(A) \rightarrow L_2(A) \) why?

\( U_1(A) \) happened strictly before \( L_2(A) \)
Theorem: 2PL ensures conflict serializability

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

Then there is the following \textit{temporal} cycle in the schedule: \( U_1(A) \rightarrow L_2(A) \) why?
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:

\[ U_1(A) \rightarrow L_2(A) \]
\[ L_2(A) \rightarrow U_2(B) \]

why?

L_2(A) happened strictly before U_1(A)
**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:

- $U_1(A) \rightarrow L_2(A)$
- $L_2(A) \rightarrow U_2(B)$

**why?**
**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:

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

**why?**
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:

\[ U_1(A) \rightarrow L_2(A) \]
\[ L_2(A) \rightarrow U_2(B) \]
\[ U_2(B) \rightarrow L_3(B) \]

......etc.....
**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:

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

**Cycle in time:** 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)</td>
<td>$L_2(A)$; READ(A)</td>
</tr>
<tr>
<td>A := A + 100</td>
<td>A := A * 2</td>
</tr>
<tr>
<td>WRITE(A); $U_1(A)$</td>
<td>WRITE(A);</td>
</tr>
<tr>
<td>READ(B)</td>
<td>$L_2(B)$; BLOCKED…</td>
</tr>
<tr>
<td>B := B + 100</td>
<td>…GRANTED; READ(B)</td>
</tr>
<tr>
<td>WRITE(B); $U_1(B)$</td>
<td>B := B * 2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B); $U_2(A)$; $U_2(B)$;</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

Rollback
A NEW PROBLEM:
NON-RECOVERABLE SCHEDULE

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>[L_1(A); \quad L_1(B); \quad \text{READ}(A)]</td>
<td>[L_2(A); \quad \text{READ}(A)]</td>
</tr>
<tr>
<td>[A := A+100]</td>
<td>[A := A*2]</td>
</tr>
<tr>
<td>[\text{WRITE}(A); \quad U_1(A)]</td>
<td>[\text{WRITE}(A);]</td>
</tr>
<tr>
<td>[\text{READ}(B)]</td>
<td>[\text{L}_2(B); \quad \text{BLOCKED}\ldots]</td>
</tr>
<tr>
<td>[B := B+100]</td>
<td>[\ldots\text{GRANTED}; \quad \text{READ}(B)]</td>
</tr>
<tr>
<td>[\text{WRITE}(B); \quad U_1(B)]</td>
<td>[B := B*2]</td>
</tr>
</tbody>
</table>

Elements A, B written by T1 are restored to their original value.

Rollback
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); , \text{READ}(A)]</td>
<td>[L_2(A); , \text{READ}(A)]</td>
</tr>
<tr>
<td>[A := A + 100]</td>
<td>[A := A \times 2]</td>
</tr>
<tr>
<td>[\text{WRITE}(A); , U_1(A)]</td>
<td>[\text{WRITE}(A);]</td>
</tr>
</tbody>
</table>

Rollback

Elements A, B written by T1 are restored to their original value.

Dirty reads of A, B lead to incorrect writes.

...GRANTED; \[\text{READ}(B)\]

\[B := B \times 2\]

\[\text{WRITE}(B); \, U_2(A); \, U_2(B);\]

Commit
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))</td>
<td>(L_2(A); READ(A))</td>
</tr>
<tr>
<td>(A := A + 100)</td>
<td>(A := A \times 2)</td>
</tr>
<tr>
<td>WRITE(A); (U_1(A))</td>
<td>WRITE(A); (U_1(A))</td>
</tr>
<tr>
<td>READ(B)</td>
<td>L(_2(\text{B}); \text{BLOCKED})…</td>
</tr>
<tr>
<td>(B := B + 100)</td>
<td>(B := B \times 2)</td>
</tr>
<tr>
<td>WRITE(B); (U_1(B))</td>
<td>WRITE(B); (U_2(A); U_2(B)); Commit</td>
</tr>
</tbody>
</table>

Elements A, B written by T1 are restored to their original value.

Dirty reads of A, B lead to incorrect writes.

Can no longer undo!
The Strict 2PL rule:

All locks are held until commit/abort:  
All unlocks are done together with commit/abort.

With strict 2PL, we will get schedules that  
are both conflict-serializable and recoverable
**STRICT 2PL**

**T1**

L₁(A); READ(A)
A := A + 100
WRITE(A);

L₁(B); READ(B)
B := B + 100
WRITE(B);

Rollback & U₁(A); U₁(B);

**T2**

L₂(A); BLOCKED…

…GRANTED; READ(A)
A := A * 2
WRITE(A);

L₂(B); READ(B)
B := B * 2
WRITE(B);

Commit & U₂(A); U₂(B);
STRICT 2PL

Lock-based systems always use strict 2PL

Easy to implement:

- Before a transaction reads or writes an element $A$, insert an L($A$)
- When the transaction commits/aborts, then release all locks

Ensures both conflict serializability and recoverability
ANOTHER PROBLEM: DEADLOCKS

$T_1$: R(A), W(B)

$T_2$: R(B), W(A)

$T_1$ holds the lock on A, waits for B

$T_2$ holds the lock on B, waits for A

This is a deadlock!
ANOTHER PROBLEM: DEADLOCKS

To detect a deadlocks, search for a cycle in the waits-for graph:

$T_1$ waits for a lock held by $T_2$;
$T_2$ waits for a lock held by $T_3$;

... 

$T_n$ waits for a lock held by $T_1$  

Relatively expensive: check periodically, if deadlock is found, then abort one TXN; re-check for deadlock more often (why?)