**ADMINISTRIVIA**

**Homework #4** will be released on Wednesday. It is due Sun Nov 7th @ 11:59pm.

**Project #3** is due Sun Nov 14th @ 11:59pm.

**Project #2** practice submission available on Gradescope.
UPCOMING DATABASE TALK

An Overview of the Starburst Trino Query Optimizer

→ Today Oct 25th @ 4:30pm ET
A DBMS's concurrency control and recovery components permeate throughout the design of its entire architecture.
A DBMS's concurrency control and recovery components permeate throughout the design of its entire architecture.
We both change the same record in a table at the same time. 

How to avoid race condition?

You transfer $100 between bank accounts but there is a power failure. 

What is the correct database state?
We both change the same record in a table at the same time. 

*How to avoid race condition?*

You transfer $100 between bank accounts but there is a power failure. 

*What is the correct database state?*
Valuable properties of DBMSs.
Based on concept of transactions with ACID properties.

Let's talk about transactions...
A **transaction** is the execution of a sequence of one or more operations (e.g., SQL queries) on a database to perform some higher-level function.

It is the basic unit of change in a DBMS: → Partial transactions are not allowed!
Move $100 from Lin' bank account to his promoter's account.

Transaction:
→ Check whether Lin has $100.
→ Deduct $100 from his account.
→ Add $100 to his promoter account.
STRAWMAN SYSTEM

Execute each txn one-by-one (i.e., serial order) as they arrive at the DBMS.
→ One and only one txn can be running at the same time in the DBMS.

Before a txn starts, copy the entire database to a new file and make all changes to that file.
→ If the txn completes successfully, overwrite the original file with the new one.
→ If the txn fails, just remove the dirty copy.
A (potentially) better approach is to allow concurrent execution of independent transactions.

Why do we want that?
A (potentially) better approach is to allow concurrent execution of independent transactions.

Why do we want that?

→ Better utilization/throughput
→ Increased response times to users.
A (potentially) better approach is to allow concurrent execution of independent transactions.

**Why do we want that?**

→ Better utilization/throughput
→ Increased response times to users.

But we also would like:

→ Correctness
→ Fairness
Arbitrary interleaving of operations can lead to:

→ Temporary Inconsistency (ok, unavoidable)
→ Permanent Inconsistency (bad!)

We need formal correctness criteria to determine whether an interleaving is valid.
A txn may carry out many operations on the data retrieved from the database.

The DBMS is only concerned about what data is read/written from/to the database.

→ Changes to the "outside world" are beyond the scope of the DBMS.
**FORMAL DEFINITIONS**

**Database:** A fixed set of named data objects (e.g., A, B, C, …).
→ We do not need to define what these objects are now.

**Transaction:** A sequence of read and write operations (R(A), W(B), …)
→ DBMS's abstract view of a user program
A newtxn starts with the **BEGIN** command.

The txn stops with either **COMMIT** or **ABORT**:
→ If commit, the DBMS either saves all the txn's changes or aborts it.
→ If abort, all changes are undone so that it's like as if the txn never executed at all.

Abort can be either self-inflicted or caused by the DBMS.
CORRECTNESS CRITERIA: ACID

Atomicity: All actions in the txn happen, or none happen.

Consistency: If each txn is consistent and the DB starts consistent, then it ends up consistent.

Isolation: Execution of one txn is isolated from that of other txns.

Durability: If a txn commits, its effects persist.
CORRECTNESS CRITERIA: ACID

Atomicity: “all or nothing”

Consistency: “it looks correct to me”

Isolation: “as if alone”

Durability: “survive failures”
TODAY'S AGENDA

Atomicity
Consistency
Isolation
Durability
Two possible outcomes of executing a txn:
→ Commit after completing all its actions.
→ Abort (or be aborted by the DBMS) after executing some actions.

DBMS guarantees that txns are **atomic**.
→ From user's point of view: txn always either executes all its actions or executes no actions at all.
Scenario #1:
→ We take $100 out of Lin’s account but then the DBMS aborts the txn before we transfer it.

Scenario #2:
→ We take $100 out of Lin's account but then there is a power failure before we transfer it.

What should be the correct state of Lin's account after both txns abort?
MECHANISMS FOR ENSURING ATOMICITY

Approach #1: Logging
→ DBMS logs all actions so that it can undo the actions of aborted transactions.
→ Maintain undo records both in memory and on disk.
→ Think of this like the black box in airplanes...

Logging is used by almost every DBMS.
→ Audit Trail
→ Efficiency Reasons
Approach #2: Shadow Paging

→ DBMS makes copies of pages and txns make changes to those copies. Only when the txn commits is the page made visible to others.

→ Originally from System R.

Few systems do this:

→ CouchDB
→ LMDB (OpenLDAP)
Approach #2: Shadow Paging

→ DBMS makes copies of pages and txns make changes to those copies. Only when the txn commits is the page made visible to others.
→ Originally from System R.

Few systems do this:
→ CouchDB
→ LMDB (OpenLDAP)
The "world" represented by the database is logically correct. All questions asked about the data are given logically correct answers.

Database Consistency
Transaction Consistency
The database accurately models the real world and follows integrity constraints.

Transactions in the future see the effects of transactions committed in the past inside of the database.
If the database is consistent before the transaction starts (running alone), it will also be consistent after.

Transaction consistency is the application's responsibility. DBMS cannot control this.

→ We won't discuss this issue further…
ISOLATION OF TRANSACTIONS

Users submit txns, and each txn executes as if it was running by itself.
→ Easier programming model to reason about.
ISOLATION OF TRANSACTIONS

Users submit txns, and each txn executes as if it was running by itself.
→ Easier programming model to reason about.

But the DBMS achieves concurrency by interleaving the actions (reads/writes of DB objects) of txns.

We need a way to interleave txns but still make it appear as if they ran one-at-a-time.
A **concurrency control** protocol is how the DBMS decides the proper interleaving of operations from multiple transactions.

Two categories of protocols:

→ **Pessimistic**: Don't let problems arise in the first place.

→ **Optimistic**: Assume conflicts are rare, deal with them after they happen.
EXAMPLE

Assume at first $A$ and $B$ each have $1000$. $T_1$ transfers $100$ from $A$'s account to $B$'s. $T_2$ credits both accounts with 6% interest.

- $T_1$: BEGIN
  - $A = A - 100$
  - $B = B + 100$
  - COMMIT

- $T_2$: BEGIN
  - $A = A \times 1.06$
  - $B = B \times 1.06$
  - COMMIT
EXEMPLARY

Assume at first \( A \) and \( B \) each have $1000.

**What are the possible outcomes of running \( T_1 \) and \( T_2 \)?**

\[
\begin{array}{|l|}
\hline
\text{T}_1 & \text{T}_2 \\
\hline
\text{BEGIN} & \text{BEGIN} \\
A=A-100 & A=A*1.06 \\
B=B+100 & B=B*1.06 \\
\text{COMMIT} & \text{COMMIT} \\
\hline
\end{array}
\]
Assume at first $A$ and $B$ each have $1000.

*What are the possible outcomes of running $T_1$ and $T_2$?*

Many! But $A+B$ should be:

$\rightarrow \ 2000 \times 1.06 = 2120$

There is no guarantee that $T_1$ will execute before $T_2$ or vice-versa, if both are submitted together. But the net effect must be equivalent to these two transactions running *serially* in some order.
Example

Legal outcomes:
→ A=954, B=1166
→ A=960, B=1160

The outcome depends on whether $T_1$ executes before $T_2$ or vice versa.
EXAMPLE

Legal outcomes:

→ $A=954$, $B=1166$  $\Rightarrow A+B=2120$
→ $A=960$, $B=1160$  $\Rightarrow A+B=2120$

The outcome depends on whether $T_1$ executes before $T_2$ or vice versa.
SERIAL EXECUTION EXAMPLE

**Schedule**

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
</table>
| BEGIN  
A=A-100  
B=B+100  
COMMIT | BEGIN  
A=A*1.06  
B=B*1.06  
COMMIT |

A=954, B=1166

**Schedule**

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
</table>
| BEGIN  
A=A-100  
B=B+100  
COMMIT | BEGIN  
A=A-100  
B=B+100  
COMMIT |

A=960, B=1160
SERIAL EXECUTION EXAMPLE

\[ A=954, \; B=1166 \]

\[ A=960, \; B=1160 \]

A+B=$2120
We interleave txns to maximize concurrency.
→ Slow disk/network I/O.
→ Multi-core CPUs.

When one txn stalls because of a resource (e.g., page fault), another txn can continue executing and make forward progress.
INTERLEAVING EXAMPLE (GOOD)

Schedule

\[ \begin{array}{c|c}
T_1 & T_2 \\
\hline
\text{BEGIN} & \text{BEGIN} \\
A=A-100 & A=A\times1.06 \\
B=B+100 & B=B\times1.06 \\
\text{COMMIT} & \text{COMMIT} \\
\end{array} \]

A=954, B=1166
INTERLEAVING EXAMPLE (GOOD)

Schedule

T₁
BEGIN
A=A-100
B=B+100
COMMIT

A=954, B=1166

T₂
BEGIN
A=A*1.06
B=B*1.06
COMMIT

Schedule

T₁
BEGIN
A=A-100
B=B+100
COMMIT

A=960, B=1160

T₂
BEGIN
A=A*1.06
B=B*1.06
COMMIT
INTERLEAVING EXAMPLE (GOOD)

Schedule

\[
\begin{array}{l}
T_1 & T_2 \\
\text{BEGIN} & \text{BEGIN} \\
A=A-100 & A=A-100 \\
B=B+100 & B=B+100 \\
\text{COMMIT} & \text{COMMIT} \\
\end{array}
\]

A=954, B=1166

Schedule

\[
\begin{array}{l}
T_1 & T_2 \\
\text{BEGIN} & \text{BEGIN} \\
A=A\times1.06 & A=A\times1.06 \\
B=B\times1.06 & B=B\times1.06 \\
\text{COMMIT} & \text{COMMIT} \\
\end{array}
\]

A=960, B=1160
BEGIN
A = A - 100
B = B + 100
COMMIT

BEGIN
A = A * 1.06
B = B * 1.06
COMMIT

A = 954, B = 1166

BEGIN
A = A - 100
B = B + 100
COMMIT

BEGIN
A = A * 1.06
B = B * 1.06
COMMIT

A = 960, B = 1160

A + B = $2120
INTERLEAVING EXAMPLE (BAD)

Schedule

\begin{align*}
\begin{array}{|c|}
\hline
T_1 & T_2 \\
\hline
\text{BEGIN} & \text{BEGIN} \\
A=A-100 & A=A \times 1.06 \\
B=B+100 & B=B \times 1.06 \\
\text{COMMIT} & \text{COMMIT} \\
\hline
A=954, B=1160 & A=954, B=1166 \neq \\
& \text{or} \\
& A=960, B=1160
\end{array}
\end{align*}
The bank is missing $6!

A = 954, B = 1166
or
A = 960, B = 1160

The bank is missing $6!
### INTERLEAVING EXAMPLE (BAD)

#### Schedule

<table>
<thead>
<tr>
<th></th>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BEGIN A = A - 100</td>
<td>BEGIN A = A \times 1.06</td>
</tr>
<tr>
<td></td>
<td>B = B + 100</td>
<td>B = B \times 1.06</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

\[ A = 954, \quad B = 1160 \]

\[ A + B = $2114 \]

#### DBMS View

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

\[ A = 954, \quad B = 1160 \]

\[ A + B = $2114 \]
INTERLEAVING EXAMPLE (BAD)

Schedule

\( T_1 \)
\[\begin{array}{|c|}
\hline
\text{BEGIN} \\
A=A-100 \\
B=B+100 \\
\text{COMMIT} \\
\hline
\end{array}\]

\( T_2 \)
\[\begin{array}{|c|}
\hline
\text{BEGIN} \\
A=A\times1.06 \\
B=B\times1.06 \\
\text{COMMIT} \\
\hline
\end{array}\]

DBMS View

\( T_1 \)
\[\begin{array}{|c|}
\hline
\text{BEGIN} \\
\text{R(A)} \\
\text{W(A)} \\
\hline
\end{array}\]

\( T_2 \)
\[\begin{array}{|c|}
\hline
\text{BEGIN} \\
\text{R(A)} \\
\text{W(A)} \\
\text{R(B)} \\
\text{W(B)} \\
\text{COMMIT} \\
\hline
\end{array}\]

\[A=954, \quad B=1160\]

\[A+B=\$2114\]
CORRECTNESS

How do we judge whether a schedule is correct?
CORRECTNESS

How do we judge whether a schedule is correct?

If the schedule is equivalent to some serial execution.
FORMAL PROPERTIES OF SCHEDULES

Serial Schedule
→ A schedule that does not interleave the actions of different transactions.

Equivalent Schedules
→ For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule.
→ Doesn't matter what the arithmetic operations are!
Serializable Schedule

→ A schedule that is equivalent to some serial execution of the transactions.

If each transaction preserves consistency, every serializable schedule preserves consistency.
Serializability is a less intuitive notion of correctness compared to txn initiation time or commit order, but it provides the DBMS with additional flexibility in scheduling operations.

More flexibility means better parallelism.
CONFLICTING OPERATIONS

We need a formal notion of equivalence that can be implemented efficiently based on the notion of "conflicting" operations.

Two operations **conflict** if:
→ They are by different transactions,
→ They are on the same object and at least one of them is a write.
INTERLEAVED EXECUTION ANOMALIES

Read-Write Conflicts (R-W)
Write-Read Conflicts (W-R)
Write-Write Conflicts (W-W)
READ-WRITE CONFLICTS

Unrepeatable Reads

T₁
BEGIN
R(A)
R(A)
COMMIT

T₂
BEGIN
R(A)
W(A)
COMMIT
Unrepeatable Reads

BEGIN R(A)
COMMIT

BEGIN
R(A)
W(A)
COMMIT

T₁
R(A)
COMMIT

T₂
$10
READ-WRITE CONFLICTS

Unrepeatable Reads

T_1

BEGIN
R(A)

R(A)
COMMIT

T_2

BEGIN
R(A)
W(A)
COMMIT

$10 ← $10

$19 ← $19
Unrepeatable Reads

```
BEGIN R(A) R(A) COMMIT
BEGIN R(A) W(A) COMMIT
```

- $10$ to $T_1$
- $10$ to $T_2$
- $19$ to $T_1$
- $19$ to $T_2$
Unrepeatable Reads

READ-WRITE CONFLICTS

BEGIN R(A) R(A)

BEGIN R(A) W(A)

COMMIT

$10

$19

T_1

T_2

R(A)

$10

$19

COMMIT

COMMIT
WRITE-READ CONFLICTS

Reading Uncommitted Data ("Dirty Reads")

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>ABORT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>
**WRITE-READ CONFLICTS**

Reading Uncommitted Data ("Dirty Reads")

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

$10$
WRITE-READ CONFLICTS

Reading Uncommitted Data ("Dirty Reads")

BEGIN R(A) W(A)
ABORT

$10
$12

BEGIN R(A) W(A)
COMMIT
WRITE-READ CONFLICTS

Reading Uncommitted Data ("Dirty Reads")

T1
BEGIN
R(A)
W(A)
ABORT

T2
BEGIN
R(A)
W(A)
COMMIT

$10
$12
$12
$12
Reading Uncommitted Data ("Dirty Reads")

**WRITE-READ CONFLICTS**

- **T_1**:
  - BEGIN
  - R(A)
  - W(A)
  - ABORT

- **T_2**:
  - BEGIN
  - R(A)
  - W(A)
  - COMMIT

-Transactions with conflicting operations:
  - $10$ and $12$
  - $12$ and $14$
Reading Uncommitted Data ("Dirty Reads")

```
BEGIN
R(A)
W(A)
ABORT

T1  T2

BEGIN
R(A)
W(A)
COMMIT
```

$10  $12
$12  $14
Writing-Read Conflicts

Reading Uncommitted Data ("Dirty Reads")

Diagram:
- **T₁**:
  - Begins with `BEGIN`
  - Reads `R(A)`
  - Writes `W(A)`
  - Aborts
- **T₂**:
  - Begins with `BEGIN`
  - Reads `R(A)`
  - Writes `W(A)`
  - Commits

Transactions:
- $10 → T₁
- $12 → T₂
- $12 → $14 in T₁
- $12 → $14 in T₂
WRITE-WRITE CONFLICTS

Overwriting Uncommitted Data

T_1
BEGIN
W(A)
W(B)
COMMIT

T_2
BEGIN
W(A)
W(B)
COMMIT
WRITE-WRITE CONFLICTS

Overwriting Uncommitted Data

BEGIN W(A) W(B) COMMIT

T₁

BEGIN W(A) W(B) COMMIT

T₂

$10

Andrew

$19

Lin
WRITE-WRITE CONFLICTS

Overwriting Uncommitted Data

T1
BEGIN
W(A)
W(B)
COMMIT

T2
BEGIN
W(A)
W(B)
COMMIT

$10
Andrew
$19
Lin
Given these conflicts, we now can understand what it means for a schedule to be serializable. 
→ This is to check whether schedules are correct. 
→ This is not how to generate a correct schedule.

There are different levels of serializability:
→ Conflict Serializability
→ View Serializability
FORMAL PROPERTIES OF SCHEDULES

Given these conflicts, we now can understand what it means for a schedule to be serializable.
→ This is to check whether schedules are correct.
→ This is not how to generate a correct schedule.

There are different levels of serializability:
→ Conflict Serializability
→ View Serializability

Most DBMSs try to support this.
Given these conflicts, we now can understand what it means for a schedule to be serializable.

→ This is to check whether schedules are correct.
→ This is not how to generate a correct schedule.

There are different levels of serializability:

→ Conflict Serializability
→ View Serializability

Most DBMSs try to support this.

No DBMS can do this.
CONFLICT SERIALIZABLE SCHEDULES

Two schedules are conflict equivalent iff:
→ They involve the same actions of the same transactions, and
→ Every pair of conflicting actions is ordered the same way.

Schedule $S$ is conflict serializable if:
→ $S$ is conflict equivalent to some serial schedule.
Schedule $S$ is conflict serializable if you can transform $S$ into a serial schedule by swapping consecutive non-conflicting operations of different transactions.
CONFLICT SERIALIZABILITY INTUITION

Schedule

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</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>
CONFLICT SERIALIZABILITY INTUITION

Schedule

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</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>

TIME
CONFLICT SERIALIZABILITY INTUITION

Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</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></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

TIME

T1

T2
CONFLICT SERIALIZABILITY INTUITION

Schedule

\[
\begin{array}{c|c}
T_1 & T_2 \\
\hline
\text{BEGIN} & \text{BEGIN} \\
\text{R(A)} & \text{R(A)} \\
\text{W(A)} & \text{W(A)} \\
\text{R(B)} & \text{W(B)} \\
\text{W(B)} & \text{R(B)} \\
\text{COMMIT} & \text{COMMIT} \\
\end{array}
\]
## CONFLICT SERIALIZABILITY INTUITION

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

**Time:** T₁, T₂
CONFLICT SERIALIZABILITY INTUITION

Schedule

\[ T_1 \]
- BEGIN
- R(A)
- W(A)
- R(B)
- W(B)
- COMMIT

\[ T_2 \]
- BEGIN
- R(A)
- W(A)
- R(B)
- W(B)
- COMMIT

TIME
### CONFLICT SERIALIZABILITY INTUITION

![Diagram showing a schedule with two transactions](image)

**Schedule**

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</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>
CONFLICT SERIALIZABILITY INTUITION

Schedule

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</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>
## Conflict Serializability Intuition

The diagram illustrates a conflict in serializability using a schedule example. The schedule consists of two transactions, $T_1$ and $T_2$, each with read and write operations on two different tables, $A$ and $B$.

### Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td>COMMIT</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
<td>R(B)</td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
<td>W(B)</td>
</tr>
</tbody>
</table>

This schedule shows a conflict because $T_1$ reads $B$ before $T_2$ writes $B$, and $T_2$ writes $A$ before $T_1$ reads $A$. This conflict cannot be resolved by any serializable schedule.
CONFLICT SERIALIZABILITY INTUITION

Schedule

\[
\begin{array}{cc}
T_1 & T_2 \\
\text{BEGIN} & \text{BEGIN} \\
R(A) & R(A) \\
W(A) & W(A) \\
R(B) & R(B) \\
W(B) & W(B) \\
\text{COMMIT} & \text{COMMIT}
\end{array}
\]

Serial Schedule

\[
\begin{array}{cc}
T_1 & T_2 \\
\text{BEGIN} & \text{BEGIN} \\
R(A) & R(A) \\
W(A) & W(A) \\
R(B) & R(B) \\
W(B) & W(B) \\
\text{COMMIT} & \text{COMMIT}
\end{array}
\]
CONFLICT SERIALIZABILITY INTUITION

Schedule

T₁

BEGIN
R(A)
W(A)
COMMIT

T₂

BEGIN
R(A)
W(A)
COMMIT
CONFLICT SERIALIZABILITY INTUITION

Schedule

\[
\begin{array}{c|c}
T_1 & T_2 \\
\hline
\text{BEGIN} & \text{BEGIN} \\
\text{R(A)} & \text{R(A)} \\
\text{W(A)} & \text{W(A)} \\
\text{COMMIT} & \text{COMMIT} \\
\end{array}
\]
CONFLICT SERIALIZABILITY INTUITION

Schedule

\[ T_1 \]

BEGIN

R(A)

BEGIN

R(A)

\[ T_2 \]

W(A)

W(A)

COMMIT

COMMIT

Serial Schedule

\[ T_1 \]

BEGIN

R(A)

BEGIN

R(A)

\[ T_2 \]

W(A)

W(A)

COMMIT

COMMIT

\[ \neq \]
Swapping operations is easy when there are only two txns in the schedule. It's cumbersome when there are many txns.

*Are there any faster algorithms to figure this out other than transposing operations?*
One node per txn.

Edge from $T_i$ to $T_j$ if:
→ An operation $O_i$ of $T_i$ conflicts with an operation $O_j$ of $T_j$ and
→ $O_i$ appears earlier in the schedule than $O_j$.

Also known as a **precedence graph**.

A schedule is conflict serializable iff its dependency graph is acyclic.
Example #1

Schedule

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

Dependency Graph

T₁ → T₂
EXAMPLE #1

Schedule

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</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>

Dependency Graph

T₁

T₂
**EXAMPLE #1**

**Schedule**

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
</table>
| **BEGIN**
  R(A)
  W(A) |
| **BEGIN**
  R(A)
  W(A) |
| R(B) |
| W(B) |
| COMMIT |

**Dependency Graph**

- Node A
- Edge: T₁ → T₂
EXAMPLE #1

Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</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>

Dependency Graph

A

T1

T2

TIME
EXAMPLE #1

Schedule

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

Dependency Graph

A

B

T₁

T₂
The cycle in the graph reveals the problem. The output of $T_1$ depends on $T_2$, and vice-versa.
EXAMPLE #2 – THREESOME

Schedule

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

Dependency Graph

T₁ → T₂ → T₃
EXAMPLE #2 – THREESOME

Schedule

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

Dependency Graph

T1 -- T2 -- T3
EXAMPLE #2 – THREESOME

Schedule:

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

Dependency Graph:

T1 → T3 → B → T2

TIME

T1

T2

T3
EXAMPLE #2 – THREESOME

Schedule

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

Dependency Graph

T1 - B - T2 - T3
EXAMPLE #2 – THREESOME

Schedule

<table>
<thead>
<tr>
<th>T_1</th>
<th>T_2</th>
<th>T_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN R(A) W(A)</td>
<td>BEGIN R(B) W(B) COMMIT</td>
<td>COMMIT</td>
</tr>
<tr>
<td>R(B) W(B) COMMIT</td>
<td>BEGIN R(A) W(A)</td>
<td></td>
</tr>
</tbody>
</table>

Dependency Graph

T_1 → B → T_2
A → T_3
EXAMPLE #2 – THREESOME

Is this equivalent to a serial execution?
EXAMPLE #2 – THREESOME

Is this equivalent to a serial execution?
Yes (T₂, T₁, T₃)
→ Notice that T₃ should go after T₂, although it starts before it!
EXAMPLE #3 – INCONSISTENT ANALYSIS

Schedule

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>A = A-10</td>
<td>sum = A</td>
</tr>
<tr>
<td>W(A)</td>
<td>R(B)</td>
</tr>
<tr>
<td></td>
<td>sum += B</td>
</tr>
<tr>
<td>R(B)</td>
<td>ECHO sum</td>
</tr>
<tr>
<td>B = B+10</td>
<td>COMMIT</td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
</tr>
</tbody>
</table>

Dependency Graph

T₁

T₂

T₁

T₂
EXAMPLE #3 – INCONSISTENT ANALYSIS

Schedule

\[
\begin{array}{|l|}
\hline
\text{T}_1 & \text{T}_2 \\
\hline
\text{BEGIN} & \text{BEGIN} \\
\text{R(A)} & \text{R(A)} \\
\text{A = A - 10} & \text{sum = A} \\
\text{W(A)} & \text{R(B)} \\
\text{R(B)} & \text{sum += B} \\
\text{B = B + 10} & \text{ECHO sum} \\
\text{W(B)} & \text{COMMIT} \\
\text{COMMIT} & \text{COMMIT} \\
\hline
\end{array}
\]

Dependency Graph

\[
\begin{array}{c}
\text{T}_1 \\
\hline
\text{T}_2 \\
\end{array}
\]
EXAMPLE #3 – INCONSISTENT ANALYSIS

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Dependency Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T₁</strong></td>
<td><strong>T₁</strong></td>
</tr>
<tr>
<td>BEGIN</td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>A = A - 10</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>B = B + 10</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
</tr>
<tr>
<td><strong>T₂</strong></td>
<td><strong>T₂</strong></td>
</tr>
<tr>
<td>BEGIN</td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>sum = A</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>sum += B</td>
<td></td>
</tr>
<tr>
<td>ECHO sum</td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
</tr>
</tbody>
</table>
EXAMPLE #3 – INCONSISTENT ANALYSIS

Schedule

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>A = A - 10</td>
<td>sum = A</td>
</tr>
<tr>
<td>W(A)</td>
<td>R(B)</td>
</tr>
<tr>
<td>R(B)</td>
<td>sum += B</td>
</tr>
<tr>
<td>B = B + 10</td>
<td>ECHO sum</td>
</tr>
<tr>
<td>W(B)</td>
<td>COMMIT</td>
</tr>
<tr>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Dependency Graph

T₁

T₂

T₁

T₂
EXAMPLE #3 – INCONSISTENT ANALYSIS

Schedule

\[ \begin{array}{|l|l|}
\hline
\text{T}_1 & \text{T}_2 \\
\hline
\text{BEGIN} & \text{BEGIN} \\
\text{R(A)} & \text{R(A)} \\
\text{A = A-10} & \text{sum = A} \\
\text{W(A)} & \text{R(B)} \\
\hline
\text{R(B)} & \text{sum += B} \\
\text{B = B+10} & \text{ECHO sum} \\
\text{W(B)} & \text{COMMIT} \\
\text{COMMIT} & \text{COMMIT} \\
\hline
\end{array} \]

Dependency Graph

\[ \begin{array}{c}
\text{T}_1 \\
\text{T}_2
\end{array} \]
EXAMPLE #3 – INCONSISTENT ANALYSIS

Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A = A-10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B = B+10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time</th>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sum = A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sum += B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECHO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dependency Graph

T₁

T₂

T₁

T₂
EXAMPLE #3 – INCONSISTENT ANALYSIS

Schedule

BEGIN
R(A)
A = A-10
W(A)

R(B)
B = B+10
W(B)
COMMIT

Dependency Graph

BEGIN
R(A)
sum = A
R(B)
sum += B
ECHO sum
COMMIT

T1
T2

T1
T2

A

TIME

CMU-DB
15-445/645 (Fall 2021)
EXAMPLE #3 – INCONSISTENT ANALYSIS

Schedule

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>R(A)</td>
<td>BEGIN</td>
</tr>
<tr>
<td></td>
<td>A = A-10</td>
<td>sum = A</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sum += B</td>
</tr>
<tr>
<td>R(B)</td>
<td>B = B+10</td>
<td>ECHO sum</td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Dependency Graph

A

T1

T2
EXAMPLE #3 – INCONSISTENT ANALYSIS

Schedule

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN R(A)</td>
<td>BEGIN R(A)</td>
</tr>
<tr>
<td>A = A - 10</td>
<td>sum = A</td>
</tr>
<tr>
<td>W(A)</td>
<td>R(B)</td>
</tr>
<tr>
<td>R(B)</td>
<td>sum += B</td>
</tr>
<tr>
<td>B = B - 10</td>
<td>ECHO sum</td>
</tr>
<tr>
<td>W(B)</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Dependency Graph

A  --→  B
  |
↓   |

T₁  ---→  T₂

T₁  ---→  T₂

T₂  ---→  T₁

T₁  ---→  T₂

T₂  ---→  T₁
Is it possible to modify only the application logic so that schedule produces a "correct" result but is still not conflict serializable?
EXAMPLE #3 – INCONSISTENT ANALYSIS

Is it possible to modify only the application logic so that schedule produces a "correct" result but is still not conflict serializable?
Alternative (weaker) notion of serializability.

Schedules $S_1$ and $S_2$ are view equivalent if:

$\rightarrow$ If $T_1$ reads initial value of $A$ in $S_1$, then $T_1$ also reads initial value of $A$ in $S_2$.

$\rightarrow$ If $T_1$ reads value of $A$ written by $T_2$ in $S_1$, then $T_1$ also reads value of $A$ written by $T_2$ in $S_2$.

$\rightarrow$ If $T_1$ writes final value of $A$ in $S_1$, then $T_1$ also writes final value of $A$ in $S_2$. 
### VIEW SERIALIZABILITY

**Schedule**

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

**Dependency Graph**

- T₁
- T₂
- T₃
VIEW SERIALIZABILITY

Schedule

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

Dependency Graph

A

T₁ -- T₂

T₃
VIEW SERIALIZABILITY

Schedule

\[ T_1 \]
\[ \text{BEGIN} \]
\[ R(A) \]
\[ W(A) \]
\[ \text{COMMIT} \]
\[ T_2 \]
\[ \text{BEGIN} \]
\[ W(A) \]
\[ \text{COMMIT} \]
\[ T_3 \]
\[ \text{BEGIN} \]
\[ W(A) \]
\[ \text{COMMIT} \]

Dependency Graph

\[ T_1 \]
\[ A \]
\[ T_2 \]
\[ T_3 \]
VIEW SERIALIZABILITY

Schedule

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

Dependency Graph

- T1: A → A
- T2: A → A
- T3: A → A

TIME
VIEW SERIALIZABILITY

Schedule

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

Dependency Graph

A

T1

T2

T3
VIEW SERIALIZABILITY

Schedule

Dependancy Graph

TIME
VIEW SERIALIZABILITY

Schedule

T₁  T₂  T₃
BEGIN R(A)  BEGIN W(A)  BEGIN
W(A)  COMMIT  W(A)  COMMIT
COMMIT  COMMIT  COMMIT

T₁  T₂  T₃
BEGIN R(A)  BEGIN W(A)  BEGIN
W(A)  COMMIT  W(A)  COMMIT
COMMIT  COMMIT  COMMIT

VIEW T₁  T₂  T₃

≡

T₁  T₂  T₃
BEGIN R(A)  W(A)  COMMIT
BEGIN W(A)  COMMIT
BEGIN W(A)  COMMIT
BEGIN W(A)  COMMIT
VIEW SERIALIZABILITY

Schedule

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

Schedule

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

VIEW T₁ T₂ T₃ T₁ T₂ T₃
VIEW SERIALIZABILITY

Allows all conflict serializable schedules + "blind writes"
View Serializability allows for (slightly) more schedules than Conflict Serializability does. → But is difficult to enforce efficiently.

Neither definition allows all schedules that you would consider "serializable". → This is because they don't understand the meanings of the operations or the data (recall example #3)
In practice, **Conflict Serializability** is what systems support because it can be enforced efficiently.

To allow more concurrency, some special cases get handled separately at the application level.
UNIVERSE OF SCHEDULES

All Schedules
UNIVERSE OF SCHEDULES

All Schedules

Serial
UNIVERSE OF SCHEDULES

All Schedules

Conflict Serializable

Serial
UNIVERSE OF SCHEDULES

All Schedules

View Serializable

Conflict Serializable

Serial
All the changes of committed transactions should be persistent.
→ No torn updates.
→ No changes from failed transactions.

The DBMS can use either logging or shadow paging to ensure that all changes are durable.
ACID PROPERTIES

**Atomicity:** All actions in the txn happen, or none happen.

**Consistency:** If each txn is consistent and the DB starts consistent, then it ends up consistent.

**Isolation:** Execution of one txn is isolated from that of other txns.

**Durability:** If a txn commits, its effects persist.
Concurrency control and recovery are among the most important functions provided by a DBMS.

Concurrency control is automatic

→ System automatically inserts lock/unlock requests and schedules actions of different txns.
→ Ensures that resulting execution is equivalent to executing the txns one after the other in some order.
Concurrency control and recovery are among the most important functions provided by a DBMS. Concurrency control is automatic, meaning the system automatically inserts lock/unlock requests and schedules actions of different transactions to executing the transactions one after the other, ensuring that the resulting execution is equivalent to executing the transactions one after the other in some order.
Concurrency control and recovery are among the most important functions provided by a DBMS. Concurrency control is automatic → System automatically inserts lock requests and schedules actions of different transactions. This ensures the resulting execution is equivalent to executing the transactions one after the other in some order.

We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions. Running two-phase commit over Paxos...
Concurrency control and recovery are among the most important functions provided by a DBMS.

Concurrency control is automatic
→ System automatically inserts lock/unlock requests and schedules actions of different txns.
→ Ensures that resulting execution is equivalent to executing the txns one after the other in some order.
NEXT CLASS

Two-Phase Locking
Isolation Levels