Concurrency Control Theory
A DBMS's concurrency control and recovery components permeate throughout the design of its entire architecture.
MOTIVATION

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 shared database to perform some higher-level function.

It is the basic unit of change in a DBMS: → Partial transactions are not allowed!
TRANSACTION EXAMPLE

Move $100 from Andy’ bank account to his bookie’s account.

Transaction:
→ Check whether Andy has $100.
→ Deduct $100 from his account.
→ Add $100 to his bookie’s 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?
→ Utilization/throughput
→ Increased response times to users.

But we also would like:
→ Correctness
→ Fairness
TRANSACTIONS

Hard to ensure correctness...
→ What happens if Andy only has $100 and tries to pay off two bookies at the same time?

Hard to execute quickly...
→ What happens if Andy needs to pay off his gambling debts very quickly all at once?
**PROBLEM STATEMENT**

Arbitrary interleaving can lead to
→ Temporary inconsistency (ok, unavoidable)
→ Permanent inconsistency (bad!)

Need formal correctness criteria.
A txn may carry out many operations on the data retrieved from the database.

However, 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 new txn starts with the **BEGIN** command.

The txn stops with either **COMMIT** or **ABORT**:
- If commit, all changes are saved.
- 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
ATOMICITY OF TRANSACTIONS

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.
We take $100 out of Andy’s account but then there is a power failure before we transfer it to his bookie.

*When the database comes back on-line, what should be the correct state of Andy’s account?*
MECHANISMS FOR ENSURING ATOMICITY

Approach #1: Logging
→ DBMS logs all actions so that it can undo the actions of aborted transactions.
→ Think of this like the black box in airplanes...

Logging used by all modern systems.
→ Audit Trail & Efficiency Reasons
MECHANISMS FOR ENSURING ATOMICITY

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)
CONSISTENCY

The "world" represented by the database is logically correct. All questions asked about the data are given logically correct answers.

Database Consistency
Transaction Consistency
DATABASE 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.
TRANSACTION CONSISTENCY

If the database is consistent before the transaction starts (running alone), it will also be consistent after.

Transaction consistency is the application’s responsibility.
→ We won’t discuss this further…
ISOLATION OF TRANSACTIONS

Users submit txns, and each txn executes as if it was running by itself.

Concurrency is achieved by DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.

How do we achieve this?
MECHANISMS FOR ENSURING ISOLATION

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.

\[
\begin{align*}
&T_1 \\
&\text{BEGIN} \\
&A=A-100 \\
&B=B+100 \\
&\text{COMMIT}
\end{align*}
\]

\[
\begin{align*}
&T_2 \\
&\text{BEGIN} \\
&A=A\times1.06 \\
&B=B\times1.06 \\
&\text{COMMIT}
\end{align*}
\]
EXAMPLE

Assume at first $A$ and $B$ each have $1000.

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

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

$T_2$
BEGIN
A=A*1.06
B=B*1.06
COMMIT
EXAMPLE

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:

→ $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+B=$2120
→ A=960, B=1160 → A+B=$2120

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

\[
\begin{align*}
A &= 954, \quad B = 1166 \\
A &= 960, \quad B = 1160
\end{align*}
\]

\[
\begin{align*}
A+B &= \$2120
\end{align*}
\]
INTERLEAVING TRANSACTIONS

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

\[ \begin{align*}
T_1 & : \\
\text{BEGIN} & : A = A - 100 \\
B = B + 100 & \\
\text{COMMIT} & \\
\text{Schedule} & : A = 954, B = 1166
\end{align*} \]

\[ \begin{align*}
T_2 & : \\
\text{BEGIN} & : A = A \times 1.06 \\
B = B \times 1.06 & \\
\text{COMMIT} & \\
\text{Schedule} & : A = 960, B = 1160
\end{align*} \]
INTERLEAVING EXAMPLE (GOOD)

Schedule

\[ T_1 \]
BEGIN
A = A - 100
B = B + 100
COMMIT

\[ T_2 \]
BEGIN
A = A \times 1.06
B = B \times 1.06
COMMIT

Schedule

\[ T_1 \]
BEGIN
A = A - 100
B = B + 100
COMMIT

\[ T_2 \]
BEGIN
A = A \times 1.06
B = B \times 1.06
COMMIT

\[ A = 954, \quad B = 1166 \]
\[ A = 960, \quad B = 1160 \]

\[ A + B = \$2120 \]
INTERLEAVING EXAMPLE (BAD)

**Schedule**

- **T1**
  - BEGIN
  - A = A - 100
  - B = B + 100
  - COMMIT

- **T2**
  - BEGIN
  - A = A * 1.06
  - B = B * 1.06
  - COMMIT

**A = 954, B = 1166**

**A = 960, B = 1160**

The bank is missing $106!
INTERLEAVING EXAMPLE (BAD)

**Schedule**

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td>A = A - 100</td>
<td>A = A * 1.06</td>
</tr>
<tr>
<td></td>
<td>B = B + 100</td>
</tr>
<tr>
<td></td>
<td>B = B * 1.06, COMMIT</td>
</tr>
<tr>
<td></td>
<td>A = 954, B = 1060</td>
</tr>
</tbody>
</table>

**DBMS View**

<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>R(B)</td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
</tr>
<tr>
<td></td>
<td>A = 954, B = 1060</td>
</tr>
</tbody>
</table>

A + B = $2014
**INTERLEAVING EXAMPLE (BAD)**

**Schedule**

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td>A=A-100</td>
<td>R(A), W(A)</td>
</tr>
<tr>
<td></td>
<td>A=A*1.06</td>
</tr>
<tr>
<td></td>
<td>B=B+100</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
</tr>
<tr>
<td></td>
<td>B=B*1.06</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
</tr>
<tr>
<td>A=954, B=1060</td>
<td></td>
</tr>
</tbody>
</table>

**DBMS View**

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td>R(A), W(A)</td>
<td>R(A), W(A)</td>
</tr>
<tr>
<td>A=A*1.06</td>
<td>B=B+100</td>
</tr>
<tr>
<td></td>
<td>B=B*1.06</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
</tr>
<tr>
<td></td>
<td>R(B), W(B)</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

\[A+B=\$2014\]
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!
FORMAL PROPERTIES OF SCHEDULES

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

If each transaction preserves consistency, every serializable schedule preserves consistency.
FORMAL PROPERTIES OF SCHEDULES

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

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

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

Transaction $T_1$ reads $A$ before writing it, while transaction $T_2$ reads $A$ after writing it. This can lead to inconsistencies if the reads are not atomic.
WRITE-READ CONFLICTS

Reading Uncommitted Data ("Dirty Reads")

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

Overwriting Uncommitted Data

\[ \text{BEGIN} \]
\[ \text{W(A)} \]
\[ \text{W(B)} \]
\[ \text{COMMIT} \]

\[ \text{BEGIN} \]
\[ \text{W(A)} \]
\[ \text{W(B)} \]
\[ \text{COMMIT} \]

Andy
$19$

Bieber
$10$

$T_1$

$T_2$
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
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.*
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.
CONFLICT SERIALIZABILITY INTUITION

Schedule $S$ is conflict serializable if you are able to 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>

**TIME**

**BEGIN**

R(A)
W(A)

R(B)
W(B)

COMMIT
CONFLICT SERIALIZABILITY INTUITION

Schedule

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

TIME

Schedule

- Begin transaction T₁
  - Read A
  - Write A
  - Read B
  - Write B
  - Commit

- Begin transaction T₂
  - Read A
  - Write A
  - Read B
  - Write B
  - Commit
## Conflict Serializability Intuition

### Schedule

<table>
<thead>
<tr>
<th>TIME</th>
<th>Schedule</th>
</tr>
</thead>
</table>
| $T_1$ | BEGIN  
      | R(A)   
      | W(A)   
      | R(B)   
      | W(B)   
      | COMMIT |
| $T_2$ | BEGIN  
      | R(A)   
      | W(A)   
      | R(B)   
      | W(B)   
      | COMMIT |
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>

BEGIN
R(A)
W(A)
R(B)
W(B)
COMMIT

BEGIN
R(A)
W(A)
R(B)
W(B)
COMMIT
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></td>
</tr>
<tr>
<td>R(B)</td>
<td>W(A)</td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td>R(B)</td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
</tr>
<tr>
<td></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

Schedule:

- T1: BEGIN R(A) W(A) R(B) W(B) COMMIT
- T2: BEGIN R(A) W(A) R(B) W(B) COMMIT
CONFLICT SERIALIZABILITY INTUITION

Schedule

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

Serial Schedule

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

\[\equiv\]
CONFLICT SERIALIZABILITY INTUITION

Schedule

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

Serial Schedule

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

\[\neq\]
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>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

A

T₁

T₂
EXAMPLE #1

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>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(A) W(A) COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>
| R(B) W(B) COMMIT | R(B) | }

Dependency Graph

T_1

T_2

T_3
EXAMPLE #2 – THREESOME

**Schedule**

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

**Dependency Graph**

- T₁ → B → T₂
- T₁ → T₃

**Schedule**:
- T₁: BEGIN, R(A), W(A), COMMIT
- T₂: BEGIN, R(B), W(B), COMMIT
- T₃: BEGIN, R(A), COMIT

**Dependency Graph**:
- T₁ depends on B
- T₃ depends on T₁
EXAMPLE #2 – THREESOME

Is this equivalent to a serial execution?
Yes \((T_2, T_1, T_3)\)

→ Notice that \(T_3\) should go after \(T_2\), 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></td>
<td>ECHO sum</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Dependency Graph

- A
- T₁ → T₂
- T₂ → T₁
- T₁
- T₂

Time

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

Dependency Graph

- A
- B

T₁ → T₂
T₂ → T₁

TIME

Schedule:
- T₁: R(A), A = A - 10, W(A)
- T₂: R(A), R(B), B = B + 10, W(B), COMMIT

Dependency Graph:
- A depends on B
- B depends on A

Inconsistent analysis because T₂ reads A after T₁ has updated it.
EXAMPLE #3 – INCONSISTENT ANALYSIS

Schedule

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

Dependency Graph

A → B
B → A

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?
VIEW SERIALIZABILITY

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:

\[
\begin{array}{ccc}
T_1 & T_2 & T_3 \\
\text{BEGIN} & \text{BEGIN} & \text{BEGIN} \\
R(A) & W(A) & W(A) \\
W(A) & \text{COMMIT} & \text{COMMIT} \\
\text{COMMIT} & \text{COMMIT} & \text{COMMIT} \\
\end{array}
\]

Dependency Graph:

\[
\begin{array}{c}
T_1 \leftrightarrow A \\
T_2 \\
T_3 \\
\end{array}
\]
VIEW SERIALIZABILITY

Schedule

\[
\begin{array}{c|c|c}
\text{T}_1 & \text{T}_2 & \text{T}_3 \\
\hline
\text{BEGIN} & \text{BEGIN} & \text{BEGIN} \\
\text{R(A)} & \text{W(A)} & \text{W(A)} \\
\hline
\text{W(A)} & \text{COMMIT} & \text{COMMIT} \\
\text{COMMIT} & \text{COMMIT} & \text{COMMIT} \\
\end{array}
\]

Dependency Graph

\[
\begin{array}{c}
\text{T}_1 \\
\text{T}_2 \\
\text{T}_3 \\
\end{array}
\]

\[
\begin{array}{c}
\text{A} \\
\text{A} \\
\text{A} \\
\end{array}
\]

TIME
VIEW SERIALIZABILITY

Schedule

<table>
<thead>
<tr>
<th></th>
<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>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td>COMMIT</td>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Dependency Graph

- T₁ → A
- T₂ → A
- T₃ → 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

T₁ → A → T₂ → A → T₃ → A → T₁

TIME
VIEW SERIALIZABILITY

Schedule

<table>
<thead>
<tr>
<th>Time</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>BEGIN</td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Dependency Graph

T1 -> A -> T2
T2 -> A -> T3
T3 -> A
A -> T1

TIME
VIEW SERIALIZABILITY

Schedule

\[
\begin{array}{ccc}
T_1 & T_2 & T_3 \\
\text{BEGIN} & \text{BEGIN} & \text{BEGIN} \\
\text{R(A)} & \text{W(A)} & \text{W(A)} \\
\text{W(A)} & \text{COMMIT} & \text{COMMIT} \\
\text{COMMIT} & & \\
\end{array}
\]

Schedule

\[
\begin{array}{ccc}
T_1 & T_2 & T_3 \\
\text{BEGIN} & \text{BEGIN} & \text{BEGIN} \\
\text{R(A)} & \text{W(A)} & \text{W(A)} \\
\text{W(A)} & \text{COMMIT} & \text{COMMIT} \\
\text{COMMIT} & & \\
\end{array}
\]

TIME

\[
\begin{array}{ccc}
\text{T_1} & \text{T_2} & \text{T_3} \\
\text{BEGIN} & \text{BEGIN} & \text{BEGIN} \\
\text{R(A)} & \text{W(A)} & \text{W(A)} \\
\text{W(A)} & \text{COMMIT} & \text{COMMIT} \\
\text{COMMIT} & & \\
\end{array}
\]

\[
\begin{array}{ccc}
\text{T_1} & \text{T_2} & \text{T_3} \\
\text{BEGIN} & \text{BEGIN} & \text{BEGIN} \\
\text{R(A)} & \text{W(A)} & \text{W(A)} \\
\text{W(A)} & \text{COMMIT} & \text{COMMIT} \\
\text{COMMIT} & & \\
\end{array}
\]

Allows all conflict serializable schedules + "blind writes"
**SERIALIZABILITY**

*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

View Serializable

Conflict Serializable

Serial
TRANSACTION DURABILITY

All of 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.
**CONCLUSION**

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