Concurrency Control Theory
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?
CONCURRENCY CONTROL & RECOVERY

Valuable properties of DBMSs.
Based on concept of transactions with ACID properties.

Let’s talk about transactions...
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.
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.
PROBLEM STATEMENT

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.
DEFINITIONS

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 (A, B, C, ...)

**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 eachtxn 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.
MECHANISMS FOR ENSURING ATOMICITY

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 online, 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)
CONSISTENCY

The "world" represented by the data is correct. All questions asked about the data are correct.

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 main categories:
→ 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 B’s account to A’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
EXAMPLE

Assume at first $A$ and $B$ each have $\$1000$. What are the legal 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

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

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.
Legal outcomes:

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

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

Schedule

T₁          T₂
BEGIN
A=A+100
B=B-100
COMMIT

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

A=1166, B=954

Schedule

T₁          T₂
BEGIN
A=A*1.06
B=B*1.06
COMMIT

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

A=1160, B=960

≡

T₁

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

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

A=1166, B=954

≡

T₂

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

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

A=1160, B=960
INTERLEAVING TRANSACTIONS

We can also interleave the txns in order to maximize concurrency.
→ Slow disk/network I/O.
→ Multi-core CPUs.
BEGIN
A=A+100

B=B-100
COMMIT

A=1166, B=954

BEGIN
A=A*1.06

B=B*1.06
COMMIT

BEGIN
A=A+100

B=B-100
COMMIT

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

BEGIN
A = A \times 1.06
B = B \times 1.06
COMMIT

A = 1166, B = 954

A = 1160, B = 960

\begin{align*}
\text{A} & = 1166, \quad \text{B} = 954 \\
\text{A} & = 1160, \quad \text{B} = 960
\end{align*}
The bank lost $6!

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

or

\[ A = 1160, \ B = 960 \]
INTERLEAVING EXAMPLE (BAD)

Schedule

\[ T_1 \]

BEGIN
A=A+100

B=B-100
COMMIT

\[ T_2 \]

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

A=1166, B=960

DBMS View

\[ T_1 \]

BEGIN
R(A)
W(A)

R(B)
W(B)
COMMIT

\[ T_2 \]

BEGIN
R(A)
W(A)
R(B)
W(B)
COMMIT
INTERLEAVING EXAMPLE (BAD)

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 = 1166, B = 960

DBMS View

\[ T_1 \]
BEGIN
R(A)
W(A)

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

\[ T_1 \]
R(B)
W(B)
COMMIT
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 significant additional flexibility in scheduling operations.
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

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

$10 \leftrightarrow$
READ-WRITE CONFLICTS

Unrepeatable Reads
READ-WRITE CONFLICTS

Unrepeatable Reads

- $10 \leftarrow R(A)
- $19 \leftarrow R(A)
- $10 \rightarrow R(A)
- $19 \rightarrow R(A)

T₁

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>R(A)</th>
</tr>
</thead>
</table>

T₂

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

$10 \rightarrow$ 
$19 \rightarrow$
WRITE-READ CONFLICTS

Reading Uncommitted Data ("Dirty Reads")
WRITE-READ CONFLICTS

Reading Uncommitted Data ("Dirty Reads")

BEGIN
R(A)
W(A)

T_1

T_2

BEGIN
R(A)
W(A)

COMMIT

R(B)
W(B)
ABORT

$10
$12

$12
$14

CMU 15-445/645 (Fall 2017)
41
WRITE-READ CONFLICTS

Reading Uncommitted Data ("Dirty Reads")

T1

BEGIN
R(A)
W(A)

T2

BEGIN
R(A)
W(A)
COMMIT

R(B)
W(B)

$10 $12

$12 $14

ABORT
WRITE-WRITE CONFLICTS

Overwriting Uncommitted Data

BEGIN
W(A)

T_1

W(B)
COMMIT

$10

Bieber

T_2

BEGIN
W(A)
W(B)
COMMIT

$19

Andy
WRITE-WRITE CONFLICTS

Overwriting Uncommitted Data

BEGIN
W(A)
W(B)
COMMIT

BEGIN
W(A)
W(B)
COMMIT

T_1

$10

Bieber

$19

Andy

T_2
FORMAL PROPERTIES OF SCHEDULES

There are different levels of serializability:

Conflict Serializability

View Serializability

DBMSs try to support this.

Nobody does 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.
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

T₁  T₂

BEGIN  BEGIN
R(A)  R(A)
W(A)  W(A)

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

**Time**

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

Schedule

\[
\begin{array}{c|c}
\text{T}_1 & \text{T}_2 \\
\text{BEGIN} & \text{BEGIN} \\
\text{R}(A) & \text{R}(A) \\
\text{W}(A) & \text{W}(A) \\
\text{R}(B) & \text{R}(B) \\
\text{W}(B) & \text{W}(B) \\
\text{COMMIT} & \text{COMMIT} \\
\end{array}
\]
CONFLICT SERIALIZABILITY INTUITION

Schedule

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

Schedule

\[ \begin{array}{c|c|}
T_1 & T_2 \\
\hline
\text{BEGIN} & \text{BEGIN} \\
\text{R(A)} & \text{R(A)} \\
\text{W(A)} & \text{W(A)} \\
\text{R(B)} & \text{R(B)} \\
\text{W(B)} & \text{W(B)} \\
\text{COMMIT} & \text{COMMIT} \\
\end{array} \]
## 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></td>
<td>W(A)</td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
<td>W(B)</td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
<td>COMMIT</td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
<td>COMMIT</td>
</tr>
</tbody>
</table>
CONFLICT SERIALIZABILITY
INTUITION

Schedule

<table>
<thead>
<tr>
<th>T_1</th>
<th>T_2</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></th>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td></td>
<td>BEGIN</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>R(B)</td>
</tr>
<tr>
<td>W(B)</td>
<td></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>R(B)</td>
</tr>
<tr>
<td>W(B)</td>
<td>W(B)</td>
</tr>
<tr>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Serial 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

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

Serial Schedule

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

[Diagram showing conflict between two schedules]
SERIALIZABILITY

Are there any faster algorithms to figure this out other than transposing operations?
DEPENDENCY GRAPHS

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.
DEPENDENCY GRAPHS

A schedule is conflict serializable if and only if 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>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

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>

The cycle in the graph reveals the problem. The output of T₁ depends on T₂, and vice-versa.
EXAMPLE #2 – LOST UPDATE

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⁻¹</td>
<td>A = A⁻¹</td>
</tr>
<tr>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Dependency Graph

A

T₁ ———→ T₂
EXAMPLE #2 – LOST UPDATE

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⁻¹</td>
<td>A = A⁻¹</td>
</tr>
<tr>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Dependency Graph

T₁ → A → T₂

T₂ → A → T₁
EXAMPLE #3 – THREESOME

Schedule

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

Dependency Graph

T1 ——— B ——— T2

T3
# Example #3 - Threesome

## Schedule

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

## Dependency Graph

![Dependency Graph Diagram](image)

- T1
- T2
- T3

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

**CMU 15-445/645 (Fall 2017)**
EXAMPLE #3 – 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)</td>
<td>W(A)</td>
<td>BEGIN R(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td>R(B)</td>
<td>R(B)</td>
<td></td>
<td>BEGIN R(B)</td>
</tr>
<tr>
<td>W(B)</td>
<td>W(B) COMMIT</td>
<td></td>
<td>W(B) COMMIT</td>
</tr>
</tbody>
</table>

Dependency Graph

Is this equivalent to a serial execution?

Yes (T₂, T₁, T₃)

→ Notice that T₃ should go after T₂, although it starts before it!
EXAMPLE #4 – INCONSISTENT ANALYSIS

Schedule

\begin{center}
\begin{tabular}{|c|c|}
\hline
T_1 & T_2 \\
\hline
BEGIN & BEGIN \\
R(A) & R(A) \\
A = A - 10 & sum = A \\
W(A) & R(B) \\
& sum += B \\
R(B) & ECHO sum \\
B = B + 10 & COMMIT \\
W(B) & \\
COMMIT & COMMIT \\
\hline
\end{tabular}
\end{center}

Dependency Graph

\begin{center}
\text{SUM} \quad \text{T1} \quad \text{T2}
\end{center}
EXAMPLE #4 – 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>R(B)</td>
<td>R(B)</td>
</tr>
<tr>
<td>B = B+10</td>
<td>sum += B</td>
</tr>
<tr>
<td>W(B)</td>
<td>ECHO sum</td>
</tr>
<tr>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Dependency Graph

A

T₁

T₂

T₁

T₂

A
EXAMPLE #4 – INCONSISTENT ANALYSIS

Schedule

\[
\begin{array}{c|c|c}
T_1 & T_2 & \text{TIME} \\
\hline
\text{BEGIN} & \text{BEGIN} & \text{BEGIN} \\
\text{R(A)} & \text{R(A)} & \text{R(A)} \\
\text{A = A-10} & \text{sum = A} & \text{sum = A} \\
\text{W(A)} & \text{R(B)} & \text{R(B)} \\
\text{sum += B} & \text{sum += B} & \text{sum += B} \\
\text{R(B)} & \text{ECHO sum} & \text{ECHO sum} \\
\text{B = B-10} & \text{COMMIT} & \text{COMMIT} \\
\text{W(B)} & & \\
\text{COMMIT} & & \\
\end{array}
\]
EXAMPLE #4 – INCONSISTENT ANALYSIS

Schedule

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

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

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

Dependency Graph

Is it possible to create a schedule similar to this that is "correct" but still not conflict serializable?
### Example #4 – Inconsistent Analysis

#### Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>T&lt;sub&gt;1&lt;/sub&gt;</th>
<th>T&lt;sub&gt;2&lt;/sub&gt;</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>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>
<tr>
<td></td>
<td>if(A≥0): cnt++</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if(B≥0): cnt++</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ECHO cnt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
<td></td>
</tr>
</tbody>
</table>

#### Dependency Graph

- A
- B

Is it possible to create a schedule similar to this that is "correct" but still not conflict serializable?
VIEW SERIALIZABILITY

Alternative (weaker) notion of serializability.

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

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

→ 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 $S_2$.

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

Schedule

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

Dependency Graph

A

T_1 → A

T_2

T_3

T_1 → T_3

T_2 → A
VIEW SERIALIZABILITY

Schedule

TIME

Dependency Graph

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

T_1
T_2
T_3
A
A
A

T_1
T_2
T_3
VIEW SERIALIZABILITY

Schedule

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

Dependency Graph

T₁ → A → T₂
A → T₃

T₁ → A → T₂ → A → T₃
## VIEW SERIALIZABILITY

### Schedule

<table>
<thead>
<tr>
<th></th>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BEGIN R(A)</td>
<td>BEGIN W(A)</td>
<td>BEGIN W(A)</td>
</tr>
<tr>
<td>2</td>
<td>W(A)</td>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
<tr>
<td>3</td>
<td>COMMIT</td>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

### Dependency Graph

- T₁
- T₂
- T₃

- A
- A
- A

**Time**
VIEW SERIALIZABILITY

Schedule

Time

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

Schedule

\[
\begin{array}{c|c|c}
T_1 & T_2 & T_3 \\
\hline
\text{BEGIN} & \text{BEGIN} & \text{BEGIN} \\
\text{R(A)} & \text{W(A)} & \text{W(A)} \\
\text{W(A)} & \text{COMMIT} & \text{COMMIT} \\
\text{COMMIT} & \text{COMMIT} & \text{COMMIT} \\
\end{array}
\]
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>COMMIT</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>COMMIT</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>

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 #4)
SERIALIZABILITY

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 thetxn 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.
PROJECT #3

Task #1 – Two-Phase Locking
Task #2 – Concurrent B+tree

We define the API for you. You need to provide the method implementations.

Due Date: Monday Nov 13th

http://15445.courses.cs.cmu.edu/fall2017/project3/
PLAGIARISM WARNING

Your project implementation must be your own work.
→ You may not copy source code from other groups or the web.
→ Do not publish your implementation on Github.

Plagiarism will not be tolerated. See CMU's Policy on Academic Integrity for additional information.
NEXT CLASS

Two-Phase Locking
Isolation Levels