ADMINISTRIVIA

Project #3 is due Sun April 7, 2024 @ 11:59pm
→ Q&A Session TBD

Final Exam
→ Thu May 2, 2024 @ 05:30pm-08:30pm
→ If you need (medical-based) accommodations, let the Profs know.
→ Don’t make travel plans before the final exam.
A DBMS’s concurrency control and recovery components permeate throughout the design of its entire architecture.
Read (A);
Check (A > $25);
Pay ($25);
A = A – 25;
Write (A);
Read (A);
Check (A > $25);
Pay ($25);
A = A – 25;
Write (A);
Read (A);
Check (A > $25);
Pay ($25);
A = A – 25;
Write (A);
Read \( A \); 
Check \( A > \$25 \); 
Pay \( \$25 \); 
\( A = A - 25 \); 
Write \( A \); 

You

Bank Balance : \$100

Your Significant Other

---

TRANSACTION MANAGEMENT
TRANSACTION MANAGEMENT

Read (A);
Check (A > $25);
Pay ($25);
A = A - 25;
Write (A);
Read (A);
Check (A > $25);
Pay ($25);
A = A – 25;
Write (A);
Read (A);
Check (A > $25);
Pay ($25);
A = A – 25;
Write (A);

Bank Balance: $100

Sufficient funds?
Pay $25

Read Balance: $100

Sufficient funds?
Pay $25

Your Significant Other

Yes

You

Yes
Read (A);
Check (A > $25);
Pay ($25);
A = A – 25;
Write (A);
Read (A);
Check (A > $25);
Pay ($25);
A = A – 25;
Write (A);
Read (A);
Check (A > $25);
Pay ($25);
A = A – 25;
Write (A);
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 simultaneously 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

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
PROBLEM STATEMENT

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.
→ We will discuss how to handle inserts/deletes next week.

**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, 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 txn happen, or none happen.  
“All or nothing…”

**Consistency**  
If each txn is consistent and the DB starts consistent, then it ends up consistent.  
“It looks correct to me…”

**Isolation**  
Execution of one txn is isolated from that of other txns.  
“All by myself…”

**Durability**  
If a txn commits, its effects persist.  
“I will survive…”
CORRECTNESS CRITERIA: ACID

**Atomicity**
All actions in txn happen, or none happen. "All or nothing..."

**Consistency**
If each txn is consistent and the DB starts consistent, then it ends up consistent. "It looks correct to me..."

**Isolation**
Execution of one txn is isolated from that of other txns. "All by myself..."

**Durability**
If a txn commits, its effects persist. "I will survive..."
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.
ATOMICITY OF TRANSACTIONS

Scenario #1:
→ We take $100 out of an account, but then the DBMS aborts the txn before we transfer it.

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

What should be the correct state of the 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
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 IBM System R.

Few systems do this:

→ CouchDB
→ Tokyo Cabinet
→ LMDB (OpenLDAP)
CONSISTENCY

The database accurately models the real world.
→ SQL has methods to specify integrity constraints (e.g., key definitions, **CHECK** and **ADD CONSTRAINT**) and the DBMS will enforce them.
→ Responsibility of the Application to define these constraints.
→ DBMS ensures that all ICs are true before and after the transaction ends.

A note on Eventual Consistency.
→ A committed transaction may see inconsistent results; e.g., may not see the updates of an older committed transaction.
→ Difficult for application programmers to reason about such semantics.
→ The trend is to move away from such models.
ISOLATION OF TRANSACTIONS

Users submit txns, and each txn executes as if it were 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.

\[
\begin{align*}
T_1 & \quad \text{BEGIN} \\
& \quad \text{A=A-100} \\
& \quad \text{B=B+100} \\
& \quad \text{COMMIT} \\
T_2 & \quad \text{BEGIN} \\
& \quad \text{A=A*1.06} \\
& \quad \text{B=B*1.06} \\
& \quad \text{COMMIT}
\end{align*}
\]
EXAMPLE

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

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

$$\begin{align*}
T_1 & : \\
\text{BEGIN} & \\
A &= A - 100 \\
B &= B + 100 \\
\text{COMMIT} & \\
T_2 & : \\
\text{BEGIN} & \\
A &= A \times 1.06 \\
B &= B \times 1.06 \\
\text{COMMIT} &
\end{align*}$$
EXAMPLE

Assume at first A and B each have $1000.

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

Many! But $A+B$ should be:

$\rightarrow \quad 2000 \times 1.06 = 2120$

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

Schedule

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

A=954, B=1166

Schedule

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

A=960, B=1160
SERIAL EXECUTION EXAMPLE

Schedule

<table>
<thead>
<tr>
<th>TIME</th>
<th>Schedule 1</th>
<th>Schedule 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>BEGIN A=A-100 B=B+100 COMMIT</td>
<td>BEGIN A=A-100 B=B+100 COMMIT</td>
</tr>
<tr>
<td></td>
<td>BEGIN A=A<em>1.06 B=B</em>1.06 COMMIT</td>
<td>BEGIN A=A<em>1.06 B=B</em>1.06 COMMIT</td>
</tr>
</tbody>
</table>

A=954, B=1166

A=960, B=1160

A+B=$2120
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

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

\[ T_2 \]
\[
\text{BEGIN} \\
A = A \times 1.06 \\
B = B \times 1.06 \\
\text{COMMIT}
\]

Schedule

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

\[ T_2 \]
\[
\text{BEGIN} \\
A = A \times 1.06 \\
B = B \times 1.06 \\
\text{COMMIT}
\]

Schedule

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

\[ T_2 \]
\[
\text{BEGIN} \\
A = A \times 1.06 \\
B = B \times 1.06 \\
\text{COMMIT}
\]

\[ \equiv \]

A = 954, B = 1166

A = 960, B = 1160
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

A = 954, B = 1166

\[ T_1 \]
BEGIN
A = A - 100

B = B + 100
COMMIT

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

B = B \times 1.06
COMMIT

A = 960, B = 1160

\[ A + B = \$2120 \]
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 = 954, B = 1166
A + B = $2114

\[ \neq \]
A = 954, B = 1166
or
A = 960, B = 1160

Off by $6!
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 = 954, B = 1160

DBMS View

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

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

A + B = $2114

How do we judge whether a schedule is correct?

If the schedule is equivalent to some serial execution.
**INTERLEAVING EXAMPLE (BAD)**

**Schedule**

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

A=954, B=1160

**DBMS View**

<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>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>

A+B=$2114
INTERLEAVING EXAMPLE (BAD)

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

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

Schedule

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

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

A=954, B=1160

A+B=$2114

How do we judge whether a schedule is correct?

If the schedule is equivalent to some serial execution.

T1
T2
TIME

A=954, B=1160

A+B=$2114
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.
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.

Serializability is a less intuitive notion of correctness compared to txn initiation time or commit order, but it provides the DBMS with more 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 one of them is a write.

Interleaved Execution Anomalies
→ Read-Write Conflicts (R-W)
→ Write-Read Conflicts (W-R)
→ Write-Write Conflicts (W-W)
Unrepeatable Read: Txn gets different values when reading the same object multiple times.
READ-WRITE CONFLICTS

Unrepeatable Read: Txn gets different values when reading the same object multiple times.
READ-WRITE CONFLICTS

Unrepeatable Read: Txn gets different values when reading the same object multiple times.
Unrepeatable Read: Txn gets different values when reading the same object multiple times.
Unrepeatable Read: Txn gets different values when reading the same object multiple times.
Dirty Read: One txn reads data written by another txn that has not committed yet.
**WRITE-READ CONFLICTS**

**Dirty Read:** One txn reads data written by another txn that has not committed yet.

---

**Diagram:**

- Two transactions, $T_1$ and $T_2$, are shown.
  - $T_1$ begins with `BEGIN`, reads `R(A)`, writes `W(A)`, and aborts.
  - $T_2$ begins with `BEGIN`, reads `R(A)`, writes `W(A)`.
  - $T_2$ commits.

A transaction with a dirty read is indicated with a red arrow labeled `$10$. 

---

**Example:**

- $T_1$: `BEGIN R(A) W(A) ABORT` (Aborts due to uncommitted write)
- $T_2$: `BEGIN R(A) W(A) COMMIT` (Commits its work)
**WRITE-READ CONFLICTS**

Dirty Read: Onetxn reads data written by another txn that has not committed yet.
Dirty Read: One txn reads data written by another txn that has not committed yet.
**WRITE-READ CONFLICTS**

**Dirty Read:** One txn reads data written by another txn that has not committed yet.

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

T_1

$10
$12

T_2

BEGIN R(A)
W(A)
COMMIT

$12
$14
```
Dirty Read: One txn reads data written by another txn that has not committed yet.
Dirty Read: One txn reads data written by another txn that has not committed yet.
WRITE-WRITE CONFLICTS

**Lost Update:** One txn overwrites uncommitted data from another uncommitted txn.
**WRITE-WRITE CONFLICTS**

**Lost Update:** One txn overwrites uncommitted data from another uncommitted txn.
**WRITE-WRITE CONFLICTS**

**Lost Update:** Onetxn overwrites uncommitted data from another uncommittedtxn.

---

**Diagram:**

- **T₁:**
  - BEGIN
  - W(A)
  - W(B)
  - COMMIT

- **T₂:**
  - BEGIN
  - W(A)
  - W(B)
  - COMMIT

- **Alice:**
  - W(B)
  - COMMIT

- **Bob:**
  - W(A)
  - W(B)
  - COMMIT

- **Data:**
  - Alice: $10
  - Bob: $19
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.

No DBMS can do this.
CONFLICT SERIALIZABLE SCHEDULES

Two schedules are **conflict equivalent** iff:

→ They involve the same actions of the same transactions.
→ Every pair of conflicting actions is ordered the same way.

Schedule **S** is **conflict serializable** if:

→ **S** is conflict equivalent to some serial schedule.
→ Intuition: 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>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

I
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>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>
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>

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

| $T_2$ |
| BEGIN |
| R(A)  |
| W(A)  |
| R(B)  |
| W(B)  |
| COMMIT |
CONFLICT SERIALIZABILITY INTUITION

Schedule

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

Schedule

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

Schedule

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

Serial Schedule

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

Schedule

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

Schedule

T₁
BEGIN
R(A)
W(A)
COMMIT

T₂
BEGIN
R(A)
W(A)
COMMIT
CONFLICT SERIALIZABILITY INTUITION

Schedule

T_1

BEGIN
R(A)

W(A)

COMMIT

T_2

BEGIN
R(A)

W(A)

COMMIT

Serial Schedule

T_1

BEGIN
R(A)

W(A)

COMMIT

T_2

BEGIN
R(A)

W(A)

COMMIT
SERIALIZABILITY

Swapping operations is easy when there are only two txns in the schedule. It’s cumbersome when there are many txns.

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

One node per txn.

Edge from $T_i$ to $T_j$ if:

$\rightarrow$ An operation $O_i$ of $T_i$ conflicts with an operation $O_j$ of $T_j$ and

$\rightarrow$ $O_i$ 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) W(A)</td>
<td>R(A) W(A)</td>
</tr>
<tr>
<td>R(B) W(B)</td>
<td>R(B) 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>
<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

Time:

- T₁
- T₂
EXAMPLE #1

Schedule

T₁
BEGIN
R(A)
W(A)
R(B)
W(B)
COMMIT

T₂
BEGIN
R(A)
W(A)
R(B)
W(B)
COMMIT

Dependency Graph

A

TIME

T₁

T₂
EXAMPLE #1

Schedule

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

Dependency Graph

\[
A \\
T_1 \\
\rightarrow \\
T_2
\]
EXAMPLE #1

Schedule

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

Dependency Graph

A

B

TIME

T_1

T_2
EXAMPLE #1

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

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

**Dependency Graph**

- T₁
- T₂
- T₃

Is this equivalent to a serial execution?

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

BEGIN
R(B)
W(B)
COMMIT

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

Yes (T₂, T₁, T₃) → Notice that T₃ should go after T₂, although it starts before it!
EXAMPLE #2 – THREE TRANSACTIONS

Schedule

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

Dependency Graph

- T_1
- T_2
- T_3

Schedule: T_1 → T_2 → T_3

Notice that T_3 should go after T_2, although it starts before it!
EXAMPLE #2 – THREE TRANSACTIONS

Is this equivalent to a serial execution?

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

```
BEGIN
  R(B)
  W(B)
COMMIT
```

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

Schedule

```
T_1
  BEGIN
  R(A)
  W(A)
  R(B)
  W(B)
  COMMIT

T_2
  BEGIN
  R(B)
  W(B)
  COMMIT

T_3
  BEGIN
  R(A)
  W(A)
  COMMIT
```

Dependency Graph

```
T_1
  A

T_2

T_3
```

Yes (T_2, T_1, T_3) → Notice that T_3 should go after T_2, although it starts before it!
EXAMPLE #2 – THREE TRANSACTIONS

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

Dependency Graph

- **T_1**
- **T_2**
- **T_3**

Yes (T_2, T_1, T_3)

Notice that T_3 should go after T_2, although it starts before it!
Is this equivalent to a serial execution?

BEGIN
R(A)
W(A)

BEGIN
R(B)
W(B)
COMMIT

BEGIN
R(A)
W(A)
COMMIT

T_1
T_2
T_3

Schedule

Dependency Graph

T_1
T_2
T_3

Yes (T_2, T_1, T_3) → Notice that T_3 should go after T_2, although it starts before it!
EXAMPLE #2 – THREE TRANSACTIONS

Schedule

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

Dependency Graph

- T1
- T2
- T3

Dependency Graph:
- T1 → B
- T2 → A
- T3

Notice that T3 should go after T2, although it starts before it!
EXAMPLE #2 – THREE TRANSACTIONS

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

Dependency Graph

T₁ → T₂
A
B
T₃

Is this equivalent to a serial execution?
BEGIN
R(A)
W(A)
R(B)
W(B)
COMMIT
BEGIN
R(B)
W(B)
COMMIT

Yes (T₂, T₁, T₃)

→ Notice that T₃ should go after T₂, although it starts before it!
Is this equivalent to a serial execution?

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

**Dependency Graph**

```
T₁ -- B -- T₂
    |      |
    |      |
    |      |
     A   T₃
```
EXAMPLE #2 – THREE TRANSACTIONS

Is this equivalent to a serial execution?

Schedule

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

Dependency Graph

Yes \((T_2, T_1, T_3)\)

→ Notice that \(T_3\) should go after \(T_2\), although it starts before it!
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 (broader) 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>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>COMMIT</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 COMMIT</td>
</tr>
<tr>
<td>W(A) COMMIT</td>
<td>W(A) COMMIT</td>
<td>W(A) COMMIT</td>
</tr>
</tbody>
</table>

Dependency Graph

- T₁ → 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>R(A)</td>
<td>BEGIN</td>
<td>W(A)</td>
<td>BEGIN</td>
</tr>
<tr>
<td>W(A)</td>
<td>COMMIT</td>
<td>COMMIT</td>
<td>W(A)</td>
</tr>
</tbody>
</table>

Dependency Graph

A -> T₁ -> T₂ -> T₃
VIEW SERIALIZABILITY

Schedule

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

Dependency Graph

TIME

T₁  T₂  T₃
A  A  A
T₁  A  T₂
T₂  T₃
T₃
VIEW SERIALIZABILITY

Schedule

T_1  T_2  T_3
BEGIN R(A)  BEGIN W(A)  BEGIN
W(A)  W(A)  W(A)
COMMIT  COMMIT  COMMIT

Dependency Graph

A

T_1  A

T_2  A

T_3
VIEW SERIALIZABILITY

Schedule

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

Dependency Graph

A

T1 → A → T2

A

T3 → A → T1

T2 → A → T3

T3 → A → T1
VIEW SERIALIZABILITY

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></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>
<tr>
<td></td>
<td>COMMIT</td>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

#### 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>BEGIN W(A)</td>
<td>BEGIN W(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>
VIEW SERIALIZABILITY

Schedule

T₁
BEGIN
R(A)
W(A)
COMMIT

T₂
BEGIN
W(A)
COMMIT

T₃
BEGIN
W(A)
COMMIT

Schedule

T₁
BEGIN
R(A)
W(A)
COMMIT

T₂
BEGIN
W(A)
COMMIT

T₃
BEGIN
W(A)
COMMIT

VIEW

Allows all conflict serializable schedules + “blind writes”
SERIALIZABILITY

View Serializability allows for (slightly) more schedules than Conflict Serializability does.

→ But it 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)
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 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.
**CORRECTNESS CRITERIA: ACID**

**Atomicity**  
All actions intxn happen, or none happen.  
“All or nothing...”

**Consistency**  
If each txn is consistent and the DB starts consistent, then it ends up consistent.  
“It looks correct to me...”

**Isolation**  
Execution of one txn is isolated from that of other txns.  
“All by myself...”

**Durability**  
If a txn commits, its effects persist.  
“I will survive...”
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.
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.

Spinner: Google’s Globally-Distributed Database

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
Consistency Models

This clickable map (adapted from Baitis, Davidson, Fekete et al and Viotti & Yuhalic) shows the relationships between common consistency models for concurrent systems. Arrows show the relationship between consistency models. For instance, strict serializable implies both serializability and linearity, linearity implies sequential consistency, and so on. Colors show how available each model is, for a distributed system on an asynchronous network.

https://jepsen.io/consistency
PROJECT #3 – QUERY EXECUTION

You will add support for executing queries in BusTub.

BusTub now supports (basic) SQL with a rule-based optimizer for converting AST into physical plans.

https://15445.courses.cs.cmu.edu/fall2023/project3/
PROJECT #3 – QUERY EXECUTION

SQL

Parser
Binder
Optimizer
Planner

Aggregation
Scan
Join
...

Project 3
Query Execution

Project 3
Query Processing (SQL)

Project 4
Concurrency Control

Project 4
Transaction Manager

Project 2
Index

Project 1
Buffer Pool Manager

Disk Manager

Table Heap
PROJECT #3 – TASKS

Plan Node Executors

→ Access Methods: Sequential Scan, Index Scan
→ Modifications: Insert, Delete, Update
→ Joins: Nest Loop Join, Hash Join
→ Miscellaneous: Window Aggregation, Aggregation, Limit, Sort, Top-k.

Optimizer Rule:

→ Convert a query with ORDER BY + LIMIT into a Top-k plan node.
→ Convert Nested Loops to Hash Join
→ Convert Sequential Scan to Index Scan
PROJECT #3 - LEADERBOARD

The leaderboard requires you to add additional rules to the optimizer to generate query plans.

→ It will be impossible to get a top ranking by just having the fastest implementations in Project #1 + Project #2.

Tasks:

→ Window Aggregation to Top-k
→ Column Pruning
→ More Aggressive Predicate Pushdown
→ Bloom Filter for Hash Join
DEVELOPMENT HINTS

Implement the **Insert** and **Sequential Scan** executors first so that you can populate tables and read from it.

Follow the **Project Road Map** rather than the order of the writeup.

You do **not** need to worry about transactions.

The aggregation hash table does **not** need to be backed by your buffer pool (i.e., use STL)

Gradescope is for meant for grading, **not** debugging. Write your own local tests.
THINGS TO NOTE

Do **not** change any file other than the ones that you submit to Gradescope.

Make sure you pull in the latest changes from the BusTub main branch.

Post your questions on Piazza or come to TA office hours.

Compare against our **solution in your browser**!
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.
Two-Phase Locking
Isolation Levels