Intro to Database Systems (15-445/645)

15

Concurrency Control Theory

FALL 2022

Andy Pavlo

Carnegie Mellon University
ADMINISTRIVIA

**Mid-Term Exam** is available for review with solutions during in my office.  
→ Bring your CMU ID card.

**Project #3** is due **Sunday Nov 13\textsuperscript{th} @ 11:59pm**.  
→ Q&A Session on **Tuesday Nov 1\textsuperscript{st} @ 8:00pm**

**Final Exam** is **Friday Dec 16\textsuperscript{th} @ 1:00pm**.
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?

Lost Updates
Concurrency Control

Durability
Recovery
CONCURRENCY CONTROL & RECOVERY

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 Andy's 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 eachtxn one-by-one (i.e., serial order) as they arrive at the DBMS.
→ One and only onetxn 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?**
- 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..."
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 Andy's account but then the DBMS aborts the txn before we transfer it.

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

What should be the correct state of Andy'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
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)
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.

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**.
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.

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

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

Assume at first A and B each have $1000.

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

$T_1$
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:
$\rightarrow$ $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+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

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

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

A=954, B=1166

Schedule

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

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

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

\[ T_1 \]

BEGIN
A=A-100

B=B+100

COMMIT

\[ T_2 \]

BEGIN
A=A*1.06

B=B*1.06

COMMIT

\[ \equiv \]

\[ T_1 \]

BEGIN
A=A-100

B=B+100

COMMIT

\[ T_2 \]

BEGIN
A=A*1.06

B=B*1.06

COMMIT

Schedule

A=954, B=1166

A=960, B=1160

A+B=$2120
INTERLEAVING EXAMPLE (BAD)

Schedule

\[
\begin{array}{|c|c|}
\hline
T_1 & T_2 \\
\hline
\text{BEGIN} & \text{BEGIN} \\
A=A-100 & A=A \times 1.06 \\
B=B+100 & B=B \times 1.06 \\
\text{COMMIT} & \text{COMMIT} \\
\hline
\end{array}
\]

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

A+B=$2114

The bank is missing $6!

\[\neq\]

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

or

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

Schedule

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

\[ \begin{align*}
T_2 & \quad \text{BEGIN} \\
\quad & \quad A = A \times 1.06 \\
\quad & \quad B = B \times 1.06 \\
\quad & \quad \text{COMMIT}
\end{align*} \]

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

DBMS View

\[ \begin{align*}
T_1 & \quad \text{BEGIN} \\
\quad & \quad R(A) \\
\quad & \quad W(A) \\
\quad & \quad \text{COMMIT}
\end{align*} \]

\[ \begin{align*}
T_2 & \quad \text{BEGIN} \\
\quad & \quad R(A) \\
\quad & \quad W(A) \\
\quad & \quad R(B) \\
\quad & \quad W(B) \\
\quad & \quad \text{COMMIT}
\end{align*} \]

\[ A + B = \$2114 \]
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 = 1160

A + B = $2114

DBMS View

T₁
BEGIN
R(A)
W(A)

T₂
BEGIN
R(A)
W(A)

R(B)
W(B)
COMMIT

A = 1514, B = 2260

TIME

A+B = $2114
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>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

How do we judge whether a schedule is correct?

If the schedule is equivalent to some serial execution.

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.
→ 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.

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.
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.

Diagram:
- T₁: BEGIN W(A) W(B) COMMIT
- T₂: BEGIN W(A) W(B) COMMIT

Money flow:
- Andy: W(B) COMMIT ($10)
- DJ Mooshoo: W(A) $19

Skull symbol indicating conflict.
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:

→ 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

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

TIME

T₁

T₂
CONFLICT SERIALIZABILITY INTUITION

Schedule

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
</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>W(B)</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

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

Serial Schedule

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

Schedule

\[
\begin{align*}
T_1 & \quad T_2 \\
\text{BEGIN} & \quad \text{BEGIN} \\
R(A) & \quad R(A) \\
W(A) & \quad W(A) \\
\text{COMMIT} & \quad \text{COMMIT} \\
\end{align*}
\]

Serial Schedule

\[
\begin{align*}
T_1 & \quad T_2 \\
\text{BEGIN} & \quad \text{BEGIN} \\
R(A) & \quad R(A) \\
W(A) & \quad W(A) \\
\text{COMMIT} & \quad \text{COMMIT} \\
\end{align*}
\]

\[\not\equiv\]
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?*
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**.

A schedule is conflict serializable iff its dependency graph is acyclic.
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></th>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>R(A)</td>
<td>R(B)</td>
<td>BEGIN</td>
</tr>
<tr>
<td>R(A)</td>
<td>W(A)</td>
<td>W(B)</td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>COMMIT</td>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

**Dependency Graph**

- T₁
- A
- T₂
- T₃

**Time**
EXAMPLE #2 – THREESOME

Schedule

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

Dependency Graph

- \( T_1 \)
- \( T_2 \)
- \( T_3 \)

\( A \)
EXAMPLE #2 – THREESOME

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
- A
- T3

Schedule timeline:

T1
- BEGIN R(A) W(A)
- R(B) W(B) COMMIT

T2
- BEGIN R(B) W(B) COMMIT

T3
- BEGIN R(A) W(A) COMMIT
- COMMIT
EXAMPLE #2 – THREESOME

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 → B
A → T₃ → T₂

Time
EXAMPLE #2 - THREESOME

Schedule

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

Dependency Graph

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

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

Dependency Graph

T₁
T₂
EXAMPLE #3 - INCONSISTENT ANALYSIS

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

Dependency Graph

T₁
T₂
### Example #3 - Inconsistent Analysis

#### Schedule

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

#### Dependency Graph

```
Graph

T1

T2
```

**Time**

The schedule shows the execution of two transactions, T1 and T2, with T1 running first. Transaction T1 reads A, updates A, and writes A. Transaction T2 reads A, updates B, and writes B, and then echoes the sum of A and B.

**Dependency Graph**

The dependency graph illustrates the relationship between the two transactions. T1 and T2 are connected, indicating that the outcome of T1 affects T2.
### 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><code>R(A)</code></td>
<td><code>BEGIN</code></td>
</tr>
<tr>
<td></td>
<td><code>A = A - 10</code></td>
<td><code>R(A)</code></td>
</tr>
<tr>
<td></td>
<td><code>W(A)</code></td>
<td><code>sum = A</code></td>
</tr>
<tr>
<td></td>
<td><code>R(B)</code></td>
<td><code>R(B)</code></td>
</tr>
<tr>
<td></td>
<td><code>B = B + 10</code></td>
<td><code>sum += B</code></td>
</tr>
<tr>
<td></td>
<td><code>W(B)</code></td>
<td><code>ECHO sum</code></td>
</tr>
<tr>
<td></td>
<td><code>COMMIT</code></td>
<td><code>COMMIT</code></td>
</tr>
</tbody>
</table>

#### Dependency Graph

- T₁
- T₂
- A

Dependency Graph:
- T₁ to A
- T₂ to A
EXAMPLE #3 – INCONSISTENT ANALYSIS

Schedule

\[
\begin{align*}
T_1 & \quad \text{BEGIN} \\
& \quad R(A) \\
& \quad A &= A-10 \\
& \quad W(A) \\
& \quad R(B) \\
& \quad B &= B+10 \\
& \quad W(B) \\
& \quad \text{COMMIT}
\end{align*}
\]

\[
\begin{align*}
T_2 & \quad \text{BEGIN} \\
& \quad R(A) \\
& \quad \text{sum} &= A \\
& \quad R(B) \\
& \quad \text{sum} &= \text{sum} + B \\
& \quad \text{ECHO} \quad \text{sum} \\
& \quad \text{COMMIT}
\end{align*}
\]

Dependency Graph

1. Is it possible to modify only the application logic so that schedule produces a "correct" result but is still not conflict serializable?
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></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>COMMIT</td>
</tr>
</tbody>
</table>

**Dependency Graph**

- T₁ → A → T₂ → A → T₃ → A → T₁
- A → T₁
- A → T₂
- A → T₃

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

Allows all conflict serializable schedules + "blind writes"
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.
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 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..."
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 transactions to avoid conflicts, ensuring that 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 can substantially improve availability, as long as they can serve for transaction failures. Spammer’s main goal is running across, at scale, replicated data, but we have also gained a great deal of time and eliminating runtime performance issues, we believe that Spammer’s design is a good candidate for improving the performance of the database.

**Full Text**

Conclusio

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 transactions to avoid conflicts, ensuring that 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 can substantially improve availability, as long as they can serve for transaction failures.

Spammer’s main goal is running across, at scale, replicated data, but we have also gained a great deal of time and eliminating runtime performance issues, we believe that Spammer’s design is a good candidate for improving the performance of the database.
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/fall2022/project3/
PROJECT #3 – TASKS

Plan Node Executors
→ Access Methods: Sequential Scan, Index Scan
→ Modifications: Insert, Delete
→ Joins: Nest Loop Join, Index Nested Loop Join
→ Miscellaneous: Aggregation, Limit, Sort

Optimizer Rule:
→ Convert a query with **ORDER BY + LIMIT** into a Top-N plan node.
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:
→ Join Reordering
→ Column Pruning
→ More Aggressive Predicate Pushdown
IMPLEMENT THE **Insert** and **Sequential Scan** executors first so that you can populate tables and read from it.

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.
NEXT CLASS

Two-Phase Locking
Isolation Levels