Lecture #15

Concurrency Control Theory
ADMINISTRIVIA

**Project #3** is due **Nov 12th @ 11:59pm.**
→ Q&A Session TBD

**Final Exam** is due **Tuesday Dec 12th @ 8:30-11:30am**
→ 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.
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);

You
Read Balance: $100
Sufficient funds?
Pay $25
New balance: $75
Bank Balance: $75!

Bank Balance: $100

You
Read Balance: $100
Sufficient funds?
Pay $25
New balance: $75
Bank Balance: $75!

Yes
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.
DEFINITIONS

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.
<table>
<thead>
<tr>
<th><strong>CORRECTNESS CRITERIA: ACID</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Atomicity</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Consistency</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Isolation</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Durability</strong></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
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.
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.

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

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

Assume at first A and B each have $1000.

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

$T_1$

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:

\[ \rightarrow \quad 2000 \times 1.06 = 2120 \]

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

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

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

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

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

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

\[ A+B=\$2120 \]
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:

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

- **T₂**
  - 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 = 1166

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*1.06
B=B*1.06
COMMIT

A=954, B=1160

A=954, B=1166

or

A=960, B=1160

A+B=$2114

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

A + B = $2114

DBMS View

\[ T_1 \]

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

\[ T_2 \]

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

How do we judge whether a schedule is correct?
If the schedule is equivalent to some serial execution.
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

A+B=$2114

**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.
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)
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.
READ-WRITE CONFLICTS

Unrepeatable Read: Txn gets different values when reading the same object multiple times.

Diagram:
- $T_1$: Begins with a read operation on $A$, followed by a commit operation.
- $T_2$: Begins with a read operation on $A$, then a write operation on $A$, and finally a commit operation.
- The diagram shows the relationships and conflicts between the transactions.
Dirty Read: One txn reads data written by another txn that has not committed yet.
Lost Update: One txn overwrites uncommitted data from another uncommitted txn.
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 \textbf{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 \textbf{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

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

Serial Schedule

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

Schedule

\[
\text{\texttt{BEGIN}} \quad \text{\texttt{R(A)}} \\
\text{\texttt{R(A)}} \quad \text{\texttt{W(A)}} \\
\text{\texttt{W(A)}} \quad \text{\texttt{COMMIT}} \\
\text{\texttt{COMMIT}}
\]

Serial Schedule

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

\[\neq\]
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?
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

The cycle in the graph reveals the problem. The output of $T_1$ depends on $T_2$, and vice-versa.
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 R(A) W(A)</td>
<td>BEGIN R(A) W(A)</td>
<td>BEGIN R(A) W(A) COMMIT</td>
</tr>
<tr>
<td>R(B) W(B) COMMIT</td>
<td>R(B) W(B)</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Dependency Graph

Yes (T₂, T₁, T₃)
→ Notice that T₃ should go after T₂, although it starts before it!
EXAMPLE #3 – INCONSISTENT ANALYSIS

Schedule

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

Dependency Graph

A

B

T1

T2

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

Dependency Graph

- T_1
- T_2
- T_3

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

Schedule

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

TIME

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)
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 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...”
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: the system automatically inserts lock/unlock requests and schedules actions of different transactions. This ensures 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 may improve availability, as long as they can survive for a few minutes. Spaghetti's main idea is managing, coordinating, and replicating data. Spaghetti's main meta-data is the SQL dictionary, a knowledge-management system with state transitions, a working set of SQL-like query languages, and a new data model that supports SQL-like data. Spaghetti is a distributed database that provides SQL-like query languages.
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.
Consistency Models

This clickable map (adapted from Bailis, Davidson, Fekete et al and Yiotti & Vukolic) 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 linearizability, linearizability implies sequential consistency, and so on. Colors show how available each model is, for a distributed system on an asynchronous network.

- **Strict Serializable**
- **Serializable**
- **Linearizable**

### Legend

- **Unavailable**: Not available during some types of network failures. Some or all nodes must pause operations in order to ensure safety.
- **Sticky Available**: Available on every non-faulty node, so long as clients only talk to the same servers, instead of switching to new ones.
- **Total Available**: Available on every non-faulty node, even when the network is completely down.

---

https://jepsen.io/consistency
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  ...

Executors

Transaction Manager

Table Heap

Index

Buffer Pool Manager

Disk Manager

Concurrency Control

Storage

Query Execution

Query Processing (SQL)

Project 3

Project 4
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
DEVELOPMENT HINTS

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