15 Concurrency Control Theory
Project 2 still ongoing
→ Due Wednesday, March 22\textsuperscript{nd}
→ Special office hours today and tomorrow 5 – 7 p.m.

Project 3 released late this week

Final exam Monday, May 1\textsuperscript{st}, 8:30 – 11:30 a.m.
LAST TIME: QUERY OPTIMIZATION

Heuristics / Rules
→ Rewrite the query to remove stupid / inefficient things.
→ These techniques may need to examine catalog, but they do not need to examine data.

Cost-based Search
→ Use a model to estimate the cost of executing a plan.
→ Enumerate multiple equivalent plans for a query and pick the one with the lowest cost.
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 conditions?*

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…
A **transaction** (txn) 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.
TRANSACTION EXAMPLE

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 each txn one-by-one (i.e., serial order) as they arrive at the DBMS.

→ One and only one txn can be running at the same time in the DBMS.

Before a txn starts, copy the entire database to a new file and make all changes to that file.

→ If the txn completes successfully, overwrite the original file with the new one.

→ If the txn fails, just remove the dirty copy.
A (potentially) better approach is to allow concurrent execution of independent transactions.

Why do we want that?
→ 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 internal inconsistency (ok, unavoidable)
→ Permanent inconsistency (bad!)

We need formal correctness criteria to determine whether an interleaving is valid.

Caveat: We’re only concerned with what’s happening inside the database: reads, writes, etc.
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 later.

**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 eachtxn is consistent and the DB starts  
|                   | consistent, then it ends up consistent.  
|                   | "It looks correct to me…"
| Isolation          | Eachtxn sees the DB as if it’s running  
|                   | alone in the DB.  
|                   | "All by myself…"
| Durability         | If a txn commits, its effects persist.  
|                   | "I will survive…" |
TODAY'S AGENDA

Atomicity
Isolation
Durability
Consistency
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.
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?
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)
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.

$T_1$:
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 $T_2$?*

$T_1$

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

$T_2$

```
BEGIN
A=A*1.06
B=B*1.06
COMMIT
```
SERIAL EXECUTION EXAMPLE

Schedule

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

A=954, B=1166

A+B=\$2120

Schedule

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

A=960, B=1160

A+B=\$2120
EXAMPLE

Assume at first A and B each have $1000.

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

→ More than one! But $A+B$ should be $2000 \times 1.06 = \$2120$

There is no guarantee that $T_1$ will execute before $T_2$ or vice-versa, if both are submitted together. But the net effect must be equivalent to these two transactions running **serially** in some order.
EXAMPLE

Legal outcomes:
→ $A=954, \; B=1166 \rightarrow A+B=2120$
→ $A=960, \; B=1160 \rightarrow A+B=2120$

The outcome depends on whether $T_1$ executes before $T_2$ or vice versa.
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)

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

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

\[A + B = \$2120\]
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

\( A = 954, \ B = 1160 \)

\( A + B = $2114 \)

\( A = 954, \ B = 1166 \) or
\( A = 960, \ B = 1160 \)

\( \not\equiv \)

The bank is missing $6!
INTERLEAVING EXAMPLE (BAD)

A schedule is correct if it is equivalent to some serial execution.

Schedule:

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

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

A = 954, B = 1160

DBMS View:

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

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

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

Serializability can be enforced efficiently based on the notion of conflicting operations.

Two operations conflict if:
→ They are by different transactions,
→ They are on the same object and $\geq 1$ 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:** Onetxn overwrites uncommitted data from another uncommitted txn.
FORMAL PROPERTIES OF Schedules

We can use these conflicts to prove that a schedule of operations is serializable.

There are different subtypes of serializability:
→ Conflict Serializability
→ View Serializability

Most DBMSs support this (or something like this).

No DBMS does 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

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

Serial Schedule

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

Schedule

T₁

BEGIN
R(A)

R(A)

W(A)

W(A)

COMMIT

COMMIT

T₂

Serial Schedule

T₁

BEGIN
R(A)

R(A)

W(A)

W(A)

COMMIT

COMMIT

T₂

≠
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.
EXAMPLE #1

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

Is this equivalent to a serial execution?

Yes: T_2, T_1, T_3

→ T_3 is after T_2 in the equivalent serial schedule, although it starts before it!
VIEW SERIALIZABILITY

Alternative (broader) notion of serializability.

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

$\rightarrow$ If $T_1$ reads initial value of $A$ in $S_1$, then $T_1$ also reads initial value of $A$ in $S_2$.
$\rightarrow$ If $T_1$ reads value of $A$ written by $T_2$ in $S_1$, then $T_1$ also reads value of $A$ written by $T_2$ in $S_2$.
$\rightarrow$ If $T_1$ writes final value of $A$ in $S_1$, then $T_1$ also writes final value of $A$ in $S_2$. 

VIEW SERIALIZABILITY

Schedule

Dependency Graph

TIME

T_1

T_2

T_3

BEGIN R(A)

BEGIN W(A)

BEGIN W(A)

COMMIT

COMMIT

COMMIT

A

A

A

A

T_1

T_2

T_3
VIEW SERIALIZABILITY

Schedule

\begin{array}{ccc}
\text{T}_1 & \text{T}_2 & \text{T}_3 \\
\text{BEGIN} & \text{BEGIN} & \text{BEGIN} \\
\text{R(A)} & \text{W(A)} & \text{W(A)} \\
\text{W(A)} & \text{COMMIT} & \text{COMMIT} \\
\text{COMMIT} & & \text{COMMIT}
\end{array}

Schedule

\begin{array}{ccc}
\text{T}_1 & \text{T}_2 & \text{T}_3 \\
\text{BEGIN} & \text{BEGIN} & \text{BEGIN} \\
\text{R(A)} & \text{W(A)} & \text{W(A)} \\
\text{W(A)} & \text{COMMIT} & \text{COMMIT} \\
\text{COMMIT} & & \text{COMMIT}
\end{array}

\[ \equiv \]

Schedule

\begin{array}{ccc}
\text{T}_1 & \text{T}_2 & \text{T}_3 \\
\text{BEGIN} & \text{BEGIN} & \text{BEGIN} \\
\text{R(A)} & \text{W(A)} & \text{W(A)} \\
\text{W(A)} & \text{COMMIT} & \text{COMMIT} \\
\text{COMMIT} & & \text{COMMIT}
\end{array}

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 serializable schedules.
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.
The database is *consistent* if it satisfies application-specific correctness constraints.

→ Implicit: Informally specified real-world constraints
→ Explicit: DBMS-enforced integrity constraints

Future transactions see the effects of past committed transactions.

A transaction is *consistent* if it takes the database from a consistent state to a consistent state.
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**
Each txn sees the DB as if it’s running alone in the DB.
"All by myself…"

**Durability**
If a txn commits, its effects persist.
"I will survive…"
Concurrent 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. This approach alleviates concurrency problems that it brings [9][10][19]. 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 is much more expensive than running them in a single node. This is especially true in high-stress situations.
NEXT CLASS

Two-Phase Locking
Isolation Levels