Concurrency Control Theory
ADMINISTRIVIA

Project #2 – C2 is due Sun Nov 1st @ 11:59pm

Project #3 will be released this week. It is due Sun Nov 22nd @ 11:59pm.

Homework #4 will be released next week. It is due Sun Nov 8th @ 11:59pm.
ADMINISTRIVIA

We will organize student-run discussion groups for projects.

Students can opt-in to be part of a small group (max 10 students) to discuss projects.
→ We will still run Moss so don't copy each other's code.
→ It is okay to share student-written tests.

If you want to volunteer to lead one, then we will send you database schwag.
UPCOMING DATABASE TALKS

MySQL Query Optimizer
→ Monday Nov 2\textsuperscript{nd} @ 5pm ET

EraDB "Magical Indexes"
→ Monday Nov 9\textsuperscript{th} @ 5pm ET

FaunaDB Serverless DBMS
→ Monday Nov 16\textsuperscript{th} @ 5pm ET
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.
MOTIVATION

We both change the same record in a table at the same time. How to avoid race condition?

You transfer $100 between bank accounts but there is a power failure. What is the correct database state?
CONCURRENCY CONTROL & RECOVERY

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

Let's talk about transactions…
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 promotor's account.

Transaction:
→ Check whether Andy has $100.
→ Deduct $100 from his account.
→ Add $100 to his promotor 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
transactions

Hard to ensure correctness...
→ What happens if Andy only has $100 and tries to pay off two promoters at the same time?

Hard to execute quickly...
→ What happens if Andy tries to pay off his gambling debts at the exact same time?
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.

**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 the txn happen, or none happen.

Consistency: If each txn is consistent and the DB starts consistent, then it ends up consistent.

Isolation: Execution of one txn is isolated from that of other txns.

Durability: If a txn commits, its effects persist.
CORRECTNESS CRITERIA: ACID

Atomicity: “all or nothing”

Consistency: “it looks correct to me”

Isolation: “as if alone”

Durability: “survive failures”
TODAY'S AGENDA

Atomicity
Consistency
Isolation
Durability
**ATOMICITY OF TRANSACTIONS**

Two possible outcomes of executing a txn:
→ Commit after completing all its actions.
→ Abort (or be aborted by the DBMS) after executing some actions.

DBMS guarantees that txns are atomic.
→ From user's point of view: txn always either executes all its actions or executes no actions at all.
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 System R.

Few systems do this:
→ CouchDB
→ 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.
TRANSACTION CONSISTENCY

If the database is consistent before the transaction starts (running alone), it will also be consistent after.

Transaction consistency is the application's responsibility. 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. T₁ transfers $100 from A's account to B's T₂ 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<sub>1</sub> and T<sub>2</sub>?**

**T<sub>1</sub>**

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

**T<sub>2</sub>**

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:
→ $A=954, \quad B=1166 \quad \Rightarrow \quad A+B=2120$
→ $A=960, \quad B=1160 \quad \Rightarrow \quad A+B=2120$

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

\begin{align*}
A &= 954, \quad B = 1166 \\
A &= 960, \quad B = 1160 \\
A + B &= 2120
\end{align*}
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

\[ \begin{array}{c|c}
  T_1 & T_2 \\
  \hline
  \text{BEGIN} & \text{BEGIN} \\
  A=A-100 & A=A-100 \\
  B=B+100 & B=B+100 \\
  \text{COMMIT} & \text{COMMIT} \\
  A=954, B=1166 & A=960, B=1160 \\
\end{array} \]

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

\[ \begin{array}{c|c}
  T_1 & T_2 \\
  \hline
  \text{BEGIN} & \text{BEGIN} \\
  A=A-100 & A=A-100 \\
  B=B+100 & B=B+100 \\
  \text{COMMIT} & \text{COMMIT} \\
  A=954, B=1166 & A=960, B=1160 \\
\end{array} \]

\[ \begin{array}{c|c}
  T_1 & T_2 \\
  \hline
  \text{BEGIN} & \text{BEGIN} \\
  A=A \times 1.06 & A=A \times 1.06 \\
  B=B \times 1.06 & B=B \times 1.06 \\
  \text{COMMIT} & \text{COMMIT} \\
\end{array} \]
INTERLEAVING EXAMPLE (GOOD)

\[
\begin{align*}
\text{Schedule} & \\
T_1 & \text{BEGIN } A=A-100 \\
& B=B+100 \\
& \text{COMMIT} \\
T_2 & \text{BEGIN } A=A\times1.06 \\
& B=B\times1.06 \\
& \text{COMMIT} \\
\end{align*}
\]

\[
\begin{align*}
A & = 954, \quad B = 1166 \\
\implies & \\
\text{Schedule} & \\
T_1 & \text{BEGIN } A=A-100 \\
& B=B+100 \\
& \text{COMMIT} \\
T_2 & \text{BEGIN } A=A\times1.06 \\
& B=B\times1.06 \\
& \text{COMMIT} \\
\end{align*}
\]

\[
\begin{align*}
A & = 960, \quad B = 1160 \\
\end{align*}
\]

\[
A+B=2120
\]
INTERLEAVING EXAMPLE (BAD)

The bank is missing $6!

A=954, B=1166
or
A=960, B=1160

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 COMMIT</td>
<td>B=B×1.06 COMMIT</td>
</tr>
</tbody>
</table>

A+B=$2114

A=954, B=1160
INTERLEAVING EXAMPLE (BAD)

**Schedule**

<table>
<thead>
<tr>
<th>T&lt;sub&gt;1&lt;/sub&gt;</th>
<th>T&lt;sub&gt;2&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN 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>

**DBMS View**

<table>
<thead>
<tr>
<th>T&lt;sub&gt;1&lt;/sub&gt;</th>
<th>T&lt;sub&gt;2&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN R(A) W(A)</td>
<td>BEGIN R(A) W(B)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></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.
→ 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 additional 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 at least one of them is a write.
INTERLEAVED EXECUTION ANOMALIES

Read-Write Conflicts (R-W)
Write-Read Conflicts (W-R)
Write-Write Conflicts (W-W)
READ-WRITE CONFLICTS

Unrepeatable Reads

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

Reading Uncommitted Data ("Dirty Reads")

Diagram:
- T1 begins with READ (A) and WRITE (A), then aborts.
- T2 begins with READ (A) and WRITE (A), then commits.
- The interaction results in conflicts between the transactions.

Key Elements:
- BEGIN
- R(A)
- W(A)
- ABORT
- COMMIT
WRITE-WRITE CONFLICTS

Overwriting Uncommitted Data

BEGIN W(A)
BEGIN W(A)
$10 W(B)
$19 W(B)
COMMIT
COMMIT

T₁

T₂

Bieber

Andy
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, and
→ Every pair of conflicting actions is ordered the same way.

Schedule $S$ is conflict serializable if:
→ $S$ is conflict equivalent to some serial schedule.
Schedule $S$ is conflict serializable if 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></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td>R(B)</td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
</tr>
</tbody>
</table>
CONFLICT SERIALIZABILITY INTUITION

Schedule

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

TIME
### Schedule

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

**CONFLICT SERIALIZABILITY INTUITION**
CONFLICT SERIALIZABILITY INTUITION

Schedule

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

Serial Schedule

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

\[=\]
CONFLICT SERIALIZABILITY INTUITION

Schedule

\[ \text{T}_1 \]
BEGIN
R(A)
W(A)
COMMIT

\[ \text{T}_2 \]
BEGIN
R(A)
W(A)
COMMIT

Serial Schedule

\[ \text{T}_1 \]
BEGIN
R(A)
W(A)
COMMIT

\[ \text{T}_2 \]
BEGIN
R(A)
W(A)
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 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:

$\rightarrow$ 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

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

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

Dependency Graph

A

\begin{align*}
T_1 & \\
T_2 & \\
\end{align*}
EXAMPLE #1

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

Dependency Graph

T₁ ─────── T₂

T₁

T₂

T₃
EXAMPLE #2 – THREESOME

Schedule

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

Dependency Graph

T_1 → T_2
B
T_3

Schedule TIMELINE

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

**Dependency Graph**

- T₁ → B
- A → T₂
- T₃ → T₂
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) W(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Begin</td>
<td>R(B) W(B)</td>
<td></td>
<td>Begin</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R(A) W(A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>COMMIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R(B) W(B)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

**Dependency Graph**

- T₁ → B
- B → T₂
- A → T₃
- T₃ → T₂

Is this equivalent to a serial execution? Yes (T₂, T₁, T₃)

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

Schedule

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

Dependency Graph

\begin{tikzpicture}
  \node (T1) at (0,0) {$T_1$};
  \node (T2) at (2,0) {$T_2$};
  \node at (1,1) {\text{Schedule}};
  \node at (1,2) {\text{Dependency Graph}};
\end{tikzpicture}
EXAMPLE #3 – INCONSISTENT ANALYSIS

Schedule

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

Dependency Graph

A

T₁ → T₂

TIME

Schedule

Dependency Graph
EXAMPLE #3 – INCONSISTENT ANALYSIS

**Schedule**

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

**Dependency Graph**

- A
- B
- T₁
- T₂
- Schedule
- Time

The schedule shows a conflict where T₁ reads A and writes to A before T₂ reads B and writes to B. This results in an inconsistent analysis.
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)
if(A≥0): cnt++
R(B)
if(B≥0): cnt++
ECHO  cnt
COMMIT

Dependency Graph

A

T₁

T₂

B

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 (weaker) notion of serializability.

Schedules $S_1$ and $S_2$ are view equivalent if:
- If $T_1$ reads initial value of $A$ in $S_1$, then $T_1$ also reads initial value of $A$ in $S_2$.
- If $T_1$ reads value of $A$ written by $T_2$ in $S_1$, then $T_1$ also reads value of $A$ written by $T_2$ in $S_2$.
- If $T_1$ writes final value of $A$ in $S_1$, then $T_1$ also writes final value of $A$ in $S_2$. 
VIEW SERIALIZABILITY

Schedule

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

Dependency Graph

T₁ → T₂ → T₃
### View Serializability

#### Schedule

<table>
<thead>
<tr>
<th>TIME</th>
<th>T&lt;sub&gt;1&lt;/sub&gt;</th>
<th>T&lt;sub&gt;2&lt;/sub&gt;</th>
<th>T&lt;sub&gt;3&lt;/sub&gt;</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>

#### Dependency Graph

- T<sub>1</sub> → T<sub>2</sub> → T<sub>3</sub>
VIEW SERIALIZABILITY

Schedule

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

Dependency Graph

\[
\begin{array}{c}
T_1 \quad A \\
T_2 \quad T_3
\end{array}
\]
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</td>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>
```

Dependency Graph

- T1
- T2
- T3
- A

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

Dependency Graph

A

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

Schedule

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

Dependency Graph

- $T_1$ 
- $T_2$ 
- $T_3$

Time
VIEW SERIALIZABILITY

Schedule

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

Schedule

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

TIME

\[\equiv\]

VIEW

Schedule

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

Allows all conflict serializable schedules + "blind writes"
SERIALIZABILITY

View Serializability allows for (slightly) more schedules than Conflict Serializability does.
→ But is difficult to enforce efficiently.

Neither definition allows all schedules that you would consider "serializable".
→ This is because they don't understand the meanings of the operations or the data (recall example #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.
ACID PROPERTIES

**Atomicity:** All actions in the txn happen, or none happen.

**Consistency:** If each txn is consistent and the DB starts consistent, then it ends up consistent.

**Isolation:** Execution of one txn is isolated from that of other txns.

**Durability:** If a txn commits, its effects persist.
CONCLUSION

Concurrency control and recovery are among the most important functions provided by a DBMS.

Concurrency control is automatic
→ System automatically inserts lock/unlock requests and schedules actions of different txns.
→ Ensures that resulting execution is equivalent to executing the txns one after the other in some order.
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 deadlocks and conflicts. An added benefit is that it simplifies writing applications, as they do not need to deal with locks and transactions explicitly. 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 cumbersome and doesn’t scale. Instead, Spanner’s two-phase commit is optimized for large-scale distributed systems and runs transparently to the application.

1 Introduction

Spanner: Google’s Globally-Distributed Database


Google Inc.

Abstract

Spanner is Google’s scalable, multi-zone, globally-distributed, and synchronously replicated database. It is the first system to distribute data at global scale and support automatically-managing distributed transactions. This paper describes how Spanner is implemented. It is a distributed, the-state-of-the-art database system that combines the benefits of a distributed database with the performance of a traditional, single-node database. Spanner’s main goal is running distributed, strongly-consistent, scalable, and transparently replicating large datasets. Spanner is a layer that is built on top of an underlying distributed transactional storage system. Spanner divides many workloads neatly into existing Bigtable-like services. We aim to address these limitations by improving the underlying transactional storage system. Spanner scales to support applications with large data sets and large numbers of users, while providing strong consistency guarantees for distributed transactions, at the same time maintaining performance. Spanner’s two-phase commit protocol is optimized for distributed systems and runs transparently to the application. Spanner is designed to support large, distributed applications and has been used to power Google’s large-scale systems.

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 cumbersome and doesn’t scale. Instead, Spanner’s two-phase commit is optimized for large-scale distributed systems and runs transparently to the application.
NEX T CL ASS

Two-Phase Locking
Isolation Levels