6.5830 Lecture 12
Transactions

10/24/2022
Lab 4 Out Today
Lab 3 Due Wednesday
Class Project Meetings This Week
Quiz 1 Back Tonight
Lab 4 Out Today
Lab 3 Due Wednesday
Class Project Meetings This Week
Quiz 1 Back Tonight
Where Are We?

• So far:
  – Studied relational model & SQL
  – Learned basic architecture of a database system
  – Studied different operator implementations
  – Looked at several data layouts
  – Saw how query optimizer works with statistics to select plans and operators

• What next:
  – *Concurrency Control and Recovery*: How to ensure correctness in the presence of modifications and failures to the database
  – Distributed and parallel query processing
  – “Advanced Topics”
Concurrency Control Key

Idea: Transactions

• Group related sequence of actions so they are “all or nothing”
  – If the system crashes, partial effects are not seen
  – Other transactions do not see partial effects

• A set of implementation techniques that provides this abstraction with good performance
ACID Properties of Transactions

• **Atomicity** – many actions look like one; “all or nothing”
• **Consistency** – database preserves invariants
• **Isolation** – concurrent actions don’t see each other’s results
• **Durability** – completed actions in effect after crash (“recoverable”)
Concurrent Programming Is Hard

- Example:
  \[\begin{array}{ll}
  \text{T1} & \text{T2} \\
  t = A & t = A \\
  t = t + 1 & t = t + 1 \\
  A = t & A = t \\
  \end{array}\]

- Looks correct!
- But maybe not if other updates to A are interleaved!
- Suppose T1 increment runs just before T2 increment
  - T1 increment will be lost

- Conventional approach: programmer adds locks
  - But must reason about other concurrent programs
Transactions Dramatically Simplify Concurrent Programming

• Concurrent actions are *serially equivalent*
  – I.e., appear to have run one after the other

• Programmer does not have to think about what is running at the same time!

• **One of the big ideas in computer science**

• Demo!
SQL Syntax

• **BEGIN TRANSACTION**
  – Followed by SQL operations that modify database

• **COMMIT**: make the effects of the transaction durable
  – After COMMIT returns database guarantees results present even after crash
  – And results are visible to other transactions

• **ABORT**: undo all effects of the transaction
This Lecture: Atomicity

• **Atomicity** – many actions like one; “all or nothing”
• In reality, actions take time!
  – To get atomicity, to prevent multiple actions from interfering with each other
  – I.e., are **Isolated**

• Will return to **Durability** in 2 lectures
  – E.g., how to *recover* a database after a crash into a state where no partial transactions are present
Consistency

• Preservation of invariants
• Usually expressed in terms of constraints
  – E.g., primary keys / foreign keys
  – Triggers
• Example: no employee makes more than their manager
• Requires ugly non-SQL syntax (e.g. PL/pgSQL)
• Often done in the application
Postgres PL/pgSQL Trigger Example

CREATE FUNCTION sal_check() RETURNS trigger AS $sal_check$

    DECLARE
    mgr_sal integer;

    BEGIN
    IF NEW.salary IS NOT NULL THEN
        SELECT INTO mgr_sal salary
            FROM emp
            JOIN manages
                ON NEW.eid = manages.eid
                AND emp.eid = manages.eid
                LIMIT 1;
        IF (mgr_sal < NEW.salary) THEN
            RAISE EXCEPTION 'employee cannot make more than manager';
        END IF;
    END IF;
    RETURN NEW;

    END;

$sal_check$ LANGUAGE plpgsql;

CREATE TRIGGER eid_sal BEFORE INSERT OR UPDATE ON emp
FOR EACH ROW EXECUTE FUNCTION sal_check();

NEW is the record being added
mgr_sal is a local variable
Query finds the salary of one manager

Check salary (if no manager, mgr_sal is NULL)

Declare that we want to call sal_check every time a record changes or is added to emp
How Can We Isolate Actions?

• Serialize execution: one transaction at a time
• Problems with this?
  – No ability to use multiple processors
  – Long running transactions *starve* others

• Goal: allow *concurrent* execution while preserving *serial equivalence*

• *Concurrency control* algorithms do this
Serializability

- An ordering of actions in concurrent transactions that is serially equivalent

<table>
<thead>
<tr>
<th>T1</th>
<th>RA</th>
<th>WA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RA</td>
<td>WA</td>
</tr>
<tr>
<td></td>
<td>RA</td>
<td>WA</td>
</tr>
<tr>
<td>T1</td>
<td>RA</td>
<td>WA</td>
</tr>
<tr>
<td></td>
<td>RA</td>
<td>WA</td>
</tr>
<tr>
<td></td>
<td>RA</td>
<td>WA</td>
</tr>
<tr>
<td></td>
<td>RA</td>
<td>WA</td>
</tr>
</tbody>
</table>

**RA**: Read A

**WA**: Write A, may depend on anything read previously

A/B are “objects” – e.g., records, disk pages, etc

Assume arbitrary application logic between reads and writes

*Serially equivalent* to T1 then T2
Serializability

• An ordering of actions in concurrent transactions that is serially equivalent

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>RA: Read A</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA</td>
<td>RA</td>
<td>WA: Write A, may depend on anything read previously</td>
</tr>
<tr>
<td>WA</td>
<td>WA</td>
<td>A/B are “objects” – e.g., records, disk pages, etc</td>
</tr>
<tr>
<td>RB</td>
<td>RB</td>
<td>Assume arbitrary application logic between reads and writes</td>
</tr>
<tr>
<td>WB</td>
<td>WB</td>
<td></td>
</tr>
</tbody>
</table>

Not serially equivalent – T2’s write to A is lost, couldn’t occur in a serial schedule
  In T1-T2, T2 should see T1’s write to A
  In T2-T1, T1 should see T2’s write to A
Testing for Serializability

Any schedule that is conflict serializable is view serializable, but not vice-versa
View Serializability

A particular ordering of instructions in a schedule S is *view equivalent* to a serial ordering S' iff:

- Every value read in S is the same value that was read by the same read in S'.
- The final write of every object is done by the same transaction T in S and S'.
- Less formally, all transactions in S “view” the same values they view in S', and the final state after the transactions run is the same.
Every value read in $S$ is the same value that was read by the same read in $S'$.

The final write of every object is done by the same transaction $T$ in $S$ and $S'$. 

$T2$ does final write in both schedules.
View Serializability Limitations

• Must test against each possible serial schedule to determine serial equivalence
  – NP-Hard!  
    *(For N concurrent transactions, there are \(2^N\) possible serial schedules)*

• No protocol to ensure view serializability as transactions run

• *Conflict serializability* addresses both points
Conflicting Operations

• Two operations are said to conflict if:
  – Both operations are on the same object
  – At least one operation is a write
  – E.g.,
    • $T_1^{WA}$ conflicts with $T_2^{RA}$, but
    • $T_1^{RA}$ does not conflict with $T_2^{RA}$

<table>
<thead>
<tr>
<th></th>
<th>T2</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2</td>
<td></td>
<td>R</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td>W</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
Conflict Serializability

A schedule is conflict serializable if it is possible to swap non-conflicting operations to derive a serial schedule.

Equivalently

For all pairs of conflicting operations \( \{O_1 \text{ in } T_1, O_2 \text{ in } T_2\} \) either

- \( O_1 \) always precedes \( O_2 \), or
- \( O_2 \) always precedes \( O_1 \).

\( T_1 \prec T_2 : \text{“} T_1 \text{ precedes } T_2 \text{”} \)
For all pairs of conflicting operations \{O1 in T1, O2 in T2\} either O1 always precedes O2, or O2 always precedes O1.

In conflict serializable schedule, can reorder non-conflicting ops to get serial schedule.
Precedence Graph

Given transactions Ti and Tj,
Create an edge from Ti $\rightarrow$ Tj if:

- Ti reads/writes some A before Tj writes A
  \[ RA_{Ti} < WA_{Tj} \text{ or } WA_{Ti} < WA_{Tj} \]
  \[ \text{or} \]
- Ti writes some A before Tj reads A
  \[ WA_{Ti} < RA_{Tj} \]

If there are cycles in this graph, schedule is not conflict serializable
Non-Serializable Example

Precedence Graph

Create an edge from \( T_i \to T_j \) if:

- \( T_i \) reads/writes some A before \( T_j \) writes A, or
  - \( RA_{T_i} \prec WA_{T_j} \) or \( WA_{T_i} \prec RA_{T_j} \)
- \( T_i \) writes some A before \( T_j \) reads A
  - \( WA_{T_i} \prec RA_{T_j} \)

Cycle!
Serializable Example

Create an edge from $T_i \rightarrow T_j$ if:

- $T_i$ reads/writes some $A$ before $T_j$ writes $A$, or
  - $RA_{T_i} < WA_{T_j}$ or $WA_{T_i} < WA_{T_j}$
- $T_i$ writes some $A$ before $T_j$ reads $A$
  - $WA_{T_i} < RA_{T_j}$

---

Precedence Graph

- $RA_{T_1} < WA_{T_2}$
- $WA_{T_1} < RA_{T_2}$
- $WA_{T_1} < WA_{T_2}$
- No Cycles!
Recap: 3 Ways to Test for Conflict Serializability

1. Check: For all pairs of conflicting operations \{O_1 \text{ in T}_1, O_2 \text{ in T}_2\} either
   a. $O_1$ always precedes $O_2$, or
   b. $O_2$ always precedes $O_1$.

2. Swap non-conflicting operations to get serial schedule

3. Build precedence graph, check for cycles
Any schedule that is conflict serializable is view serializable, but not vice-versa.
View vs Conflict Serializable

• Testing for view serializability is NP-Hard
  – Have to consider all possible orderings
• Conflict serializability used in practice
  – Not because of NP-Hardness
  – Because we have a way to enforce it as transactions run
• Example of schedule that is view serializable but not conflict serializable:

\[ \begin{array}{c|c|c|c}
T1 & T2 & T3 \\
RA & WA & WA \\
WA & RA & WA \\
RB & WB & RA \\
\end{array} \]

Blind Writes

Equivalent to T1, T2, T3

Conflict serializability does not permit this

Only happens with blind writes
• Is this schedule conflict serializable?

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td>COMMIT</td>
<td>COMMIT</td>
<td></td>
</tr>
</tbody>
</table>
• Is this schedule conflict serializable?

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RB</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td>RA</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td>COMMIT</td>
<td>COMMIT</td>
<td></td>
</tr>
</tbody>
</table>
Implementing Conflict Serializability

• Several different protocols
• Today: Two Phase Locking (2PL)
• Basic idea:
  – Acquire a shared (S) lock before each read of an object
  – Acquire an exclusive (X) lock before each write of an object
• Several transactions can hold an S lock
• Only one transaction can hold an X lock
• If a transaction cannot acquire a lock it waits (“blocks”)

Lock Compatibility Table

Conflicting operations (from def. of conflict serializability) are not compatible with each other
When to Release Locks

• After each op completes?
• Or after xaction is done with variable?
• No! Example of problem →
• T2 “sneaks in” and updates A and B before T1 updates B

This schedule is not serializable
Solution: Two Phase Locking

• A transaction cannot release any locks until it has acquired all of its locks

“The boss fight has two phases, but the second isn’t much harder than the first.”
Example, Revisited

• Rule: A transaction cannot release any locks until it has acquired all of its locks

This schedule is not serializable
Example, Revisited

• Rule: A transaction cannot release any locks until it has acquired all of its locks

• Serial schedule defined by lock points
  – Where they acquire last lock

This schedule *is* serializable
Correctness Intuition

• Once a transaction T reached its lock point:
  – T’s place in serial order is set
  – Any transactions that haven't acquired all their locks can’t take any conflicting actions until after T releases locks
    • Ordered later
  – Any transactions which already have all their locks must have completed their conflicting actions (released their locks) before T can proceed
    • Ordered earlier
Two Phase Locking (2PL) Protocol

• Before every read, acquire a shared lock

• Before every write, acquire an exclusive lock (or "upgrade") a shared to an exclusive lock

• Release locks only after last lock has been acquired, and ops on that object are finished
Refining 2PL

• Problems:
  – Deadlocks
  – Cascading Aborts

  – How do we know when we are done with all operations on an object?
Deadlocks

• Possible for Ti to hold a lock Tj needs, and vice versa

\[
\begin{array}{c}
\text{T1} \\
\text{RA} \\
\text{WA}
\end{array}
\quad \begin{array}{c}
\text{T2} \\
\text{RB} \\
\text{WB}
\end{array}
\]

\(T1\) waits for \(T2\) \(\rightarrow\) RA

\(T1\) waits for \(T2\) \(\rightarrow\) RA

\(T2\) waits for \(T1\) \(\leftarrow\) WA

\(\text{Waits-for graph}\)

\(\text{Cycle} \rightarrow \text{Deadlock}\)
Complex Deadlocks Are Possible

T1 waits for T2 → RB, WB
T2 waits for T3 → RC
T3 waits for T1 → RA, WA

Waits-for graph
Cycle → Deadlock
Resolving Deadlock

• Solution: abort one of the transactions
  – Recall: users can abort too

\[ \begin{align*}
  T1 & \rightarrow RA \\
  & \rightarrow WA \\
  & \rightarrow RB \\
  & \rightarrow WB \\
  T2 & \rightarrow RA \\
  & \rightarrow WA \\
  & \rightarrow RC \\
  & \rightarrow WC
\end{align*} \]

Equivalent to T2 - T1

Waits-for graph
Cycle $\rightarrow$ Deadlock

\[ T1 \text{ waits for } T2 \rightarrow RB \rightarrow WB \]\n
\[ T2 \text{ waits for } T3 \rightarrow RC \rightarrow WC \]
Cascading Aborts

- Problem: if T1 aborts, and T2 read something T1 wrote, T2 also needs to abort

**T1**
- Xlock A
- RA
- WA
- Xlock B
- Rel A

**T2**
- Xlock A
- RA
- WA

If T1 aborts here → T2 also needs to abort, it reads T1’s write of A

**Rel B**
- RB
- WB
- Rel B

**Rel A,B**
- Xlock B
- RB
- WB
- Rel A,B
Strict Two-Phase Locking

• Can avoid cascading aborts by holding exclusive locks until end of transaction

• Ensures that transactions never read other transaction’s uncommitted data
Strict Two-Phase Locking Protocol

- Before every read, acquire a shared lock

- Before every write, acquire an exclusive lock (or "upgrade") a shared to an exclusive lock

- Release locks only after last lock has been acquired, and ops on that object are finished
  
  - Release shared locks only after last lock has been acquired, and ops on that object are finished

- Release exclusive locks only after the transaction commits

- Ensures cascadeless-ness
Problem: When is it OK to release?

• How does DBMS know a transaction no longer needs a lock?
• Difficult, since transactions can be issued interactively
• In practice, this means that all locks held til end of transaction
• This is called *rigorous two-phase locking*
Rigorous Two-Phase Locking Protocol

• Before every read, acquire a shared lock

• Before every write, acquire an exclusive lock (or "upgrade") a shared to an exclusive lock

• Release (all) locks only after the transaction commits

• Ensures cascadelessness, and
• *Commit order = serialization order*
Next 1.5 Lectures

• Optimistic concurrency control: Another protocol to achieve conflict serializability

• Nuances that arise with locking granularity