CSE544
Data Management
Lectures 15: Transactions
Announcements

• HW5 is posted: short, sweet, due on 3/15

• Project milestones due on Friday

• Next Friday 3/12: Project Presentations!
  – 9am – 1pm (we may finish a bit earlier)
  – 11 teams
  – Each team gets 10’ presentation + 5’ discussion
  – Contest for the best presentation (stay tuned!)
Transactions

• We use database transactions everyday
  – Bank $$$ transfers
  – Online shopping
  – Signing up for classes

• Applications that talk to a DB **must** use transactions in order to keep the database consistent.
Motivating Example

Client 1:
UPDATE Budget
SET money=money-100
WHERE pid = 1

UPDATE Budget
SET money=money+60
WHERE pid = 2

UPDATE Budget
SET money=money+40
WHERE pid = 3

Client 2:
SELECT sum(money)
FROM Budget

Would like to treat each group of instructions as a unit
Transaction

**Definition**: a transaction is a sequence of updates to the database with the property that either all complete, or none completes (all-or-nothing).

```
START TRANSACTION

[SQL statements]

COMMIT or ROLLBACK (=ABORT)
```

In ad-hoc SQL: each statement = one transaction

This is referred to as autocommit

May be omitted if autocommit is off: first SQL query starts txn
Motivating Example

START TRANSACTION
UPDATE Budget
SET money=money-100
WHERE pid = 1

UPDATE Budget
SET money=money+60
WHERE pid = 2

UPDATE Budget
SET money=money+40
WHERE pid = 3
COMMIT (or ROLLBACK)

SELECT sum(money)
FROM Budget

With autocommit and without START TRANSACTION, each SQL command is a transaction
ROLLBACK

• If the app gets to a place where it can’t complete the transaction successfully, it can execute **ROLLBACK**

• This causes the system to “abort” the transaction
  – Database returns to a state without any of the changes made by the transaction

• Several reasons: user, application, system
ACID Properties

- **Atomicity**: Either all changes performed by transaction occur or none occurs.
- **Consistency**: A transaction as a whole does not violate integrity constraints.
- **Isolation**: Transactions appear to execute one after the other in sequence.
- **Durability**: If a transaction commits, its changes will survive failures.
What Could Go Wrong?

Why is it hard to provide ACID properties?

• **Concurrent** operations
  – Isolation problems
  – We saw one example earlier

• **Failures** can occur at any time
  – Atomicity and durability problems
  – Later lectures

• Transaction may need to **abort**
Concurrent Execution Problems

- **Write-read conflict: dirty read, inconsistent read**
  - A transaction reads a value written by another transaction that has not yet committed

- **Read-write conflict: unrepeatable read**
  - A transaction reads the value of the same object twice. Another transaction modifies that value in between the two reads

- **Write-write conflict: lost update**
  - Two transactions update the value of the same object. The second one to write the value overwrites the first change
A **schedule** is a sequence of interleaved actions from all transactions
### Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A, s)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>READ(B, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B, t)</td>
<td>WRITE(B, s)</td>
</tr>
</tbody>
</table>

A and B are elements in the database, t and s are variables in tx source code.
A Serial Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>WRITE(A, t)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>t := t+100</td>
</tr>
<tr>
<td>WRITE(B, t)</td>
<td>READ(B, t)</td>
</tr>
<tr>
<td>READ(A, s)</td>
<td>WRITE(A, s)</td>
</tr>
<tr>
<td>s := s*2</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B, s)</td>
<td>WRITE(B, s)</td>
</tr>
</tbody>
</table>

A = 2
B = 2

A = 102
B = 102

A = 204
B = 204
## A Serial Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, s)</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, s)</td>
<td>s := s*2</td>
</tr>
<tr>
<td>READ(B, s)</td>
<td>WRITE(B, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>t := t+100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A = 2</th>
<th>A = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>B = 2</td>
<td>B = 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A = 104</th>
<th>B = 104</th>
</tr>
</thead>
</table>

March 2, 2021
A schedule is *serializable* if it is equivalent to a serial schedule.
### A Serializable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A, s)</td>
</tr>
<tr>
<td>t := t + 100</td>
<td>s := s * 2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A, s)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(B, t)</td>
<td>READ(B, s)</td>
</tr>
<tr>
<td>t := t + 100</td>
<td>s := s * 2</td>
</tr>
<tr>
<td>WRITE(B, t)</td>
<td>WRITE(B, s)</td>
</tr>
</tbody>
</table>

This is a **serializable** schedule.

This is NOT a serial schedule.

<table>
<thead>
<tr>
<th>A = 2</th>
<th>B = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = 102</td>
<td>B = 2</td>
</tr>
<tr>
<td>A = 204</td>
<td>B = 2</td>
</tr>
<tr>
<td>A = 204</td>
<td>B = 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A = 204</th>
<th>B = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = 204</td>
<td>B = 204</td>
</tr>
<tr>
<td>A = 204</td>
<td>B = 204</td>
</tr>
</tbody>
</table>
A Non-Serializable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A,s)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>READ(B,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>WRITE(B,s)</td>
</tr>
</tbody>
</table>

Initial Values:
- A = 2
- B = 2

Updated Values:
- A = 102
- B = 2
- A = 204
- B = 2
- A = 204
- B = 4
- A = 204
- B = 104
Serializable Schedules

• The role of the scheduler is to ensure that the schedule is serializable

Q: Why not run only serial schedules? I.e. run one transaction after the other?
Serializable Schedules

• The role of the scheduler is to ensure that the schedule is serializable

Q: Why not run only serial schedules? I.e. run one transaction after the other?

A: Because of very poor throughput due to disk latency.

Lesson: main memory databases may schedule TXNs serially
Still Serializable, but…

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s + 200</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A, s)</td>
</tr>
<tr>
<td></td>
<td>READ(B, s)</td>
</tr>
<tr>
<td></td>
<td>s := s + 200</td>
</tr>
<tr>
<td></td>
<td>WRITE(B, s)</td>
</tr>
<tr>
<td></td>
<td>READ(B, t)</td>
</tr>
<tr>
<td></td>
<td>t := t+100</td>
</tr>
<tr>
<td></td>
<td>WRITE(B,t)</td>
</tr>
</tbody>
</table>

Schedule is serializable because t=t+100 and s=s+200 commute

...we don’t expect the scheduler to schedule this
To Be Practical

• Assume worst case updates:
  – Assume cannot commute actions done by transactions

• Therefore, we only care about reads and writes
  – Transaction = sequence of R(A)’s and W(A)’s

\[
\begin{align*}
T_1: & \; r_1(A); \; w_1(A); \; r_1(B); \; w_1(B) \\
T_2: & \; r_2(A); \; w_2(A); \; r_2(B); \; w_2(B)
\end{align*}
\]
Conflicts

• Write-Read – WR
• Read-Write – RW
• Write-Write – WW
Conflict Serializability

Conflicts:

Two actions by same transaction $T_i$: $r_i(X); w_i(Y)$

Two writes by $T_i, T_j$ to same element $w_i(X); w_j(X)$

Read/write by $T_i, T_j$ to same element $w_i(X); r_j(X)$
$r_i(X); w_j(X)$
Conflict Serializability

**Definition** A schedule is *conflict serializable* if it can be transformed into a serial schedule by a series of swappings of adjacent non-conflicting actions.

- Every *conflict-serializable* schedule is serializable.
- The converse is not true in general.
Conflict Serializability

Example:

\[ r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B) \]
Conflict Serializability

Example:

\[
\begin{align*}
    & r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B) \\
    \rightarrow \quad & r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)
\end{align*}
\]
Conflict Serializability

Example:

\[ r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B) \]
Conflict Serializability

Example:

\[
\begin{align*}
&\text{r}_1(\text{A}); \text{w}_1(\text{A}); \text{r}_2(\text{A}); \boxed{\text{w}_2(\text{A}); \text{r}_1(\text{B}); \text{w}_1(\text{B}); \text{r}_2(\text{B}); \text{w}_2(\text{B})} \\
&\text{r}_1(\text{A}); \text{w}_1(\text{A}); \text{r}_2(\text{A}); \text{r}_1(\text{B}); \text{w}_2(\text{A}); \text{w}_1(\text{B}); \text{r}_2(\text{B}); \text{w}_2(\text{B}) \\
&\text{r}_1(\text{A}); \text{w}_1(\text{A}); \text{r}_2(\text{A}); \text{r}_1(\text{B}); \text{w}_1(\text{B}); \text{r}_2(\text{A}); \text{w}_2(\text{A}); \text{r}_2(\text{B}); \text{w}_2(\text{B})
\end{align*}
\]
Conflict Serializability

Example:

\[
\begin{align*}
\text{r}_1(A); & \; w_1(A); \; r_2(A); \; \boxed{w_2(A)}; \; r_1(B); \; w_1(B); \; r_2(B); \; w_2(B) \\
\text{r}_1(A); & \; w_1(A); \; \boxed{r_2(A)}; \; r_1(B); \; \boxed{w_2(A)}; \; w_1(B); \; \boxed{r_2(B)}; \; w_2(B) \\
\text{r}_1(A); & \; w_1(A); \; r_1(B); \; w_1(B); \; \boxed{r_2(A)}; \; \boxed{w_2(A)}; \; r_2(B); \; w_2(B)
\end{align*}
\]
Conflict Serializability

Example:

\[
\begin{align*}
& r_1(A); w_1(A); r_2(A); \boxed{w_2(A)}; r_1(B); w_1(B); r_2(B); w_2(B) \\
& r_1(A); w_1(A); r_2(A); r_1(B); \boxed{w_2(A)}; w_1(B); r_2(B); w_2(B) \\
& r_1(A); w_1(A); r_1(B); r_2(A); \boxed{w_2(A)}; w_1(B); r_2(B); w_2(B) \\
& \boxed{r_1(A)}; w_1(A); r_1(B); w_1(B); r_2(A); \boxed{w_2(A)}; r_2(B); w_2(B) \\
& \text{...} \\
& r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)
\end{align*}
\]
### Serializable, Not Conflict-Serializable

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>READ(A, t)</strong></td>
<td><strong>READ(A, s)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>t := t + 100</strong></td>
<td><strong>s := s + 200</strong></td>
</tr>
<tr>
<td></td>
<td><strong>WRITE(A, t)</strong></td>
<td><strong>WRITE(A, s)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>READ(B, s)</strong></td>
<td><strong>READ(B, s)</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>s := s + 200</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>WRITE(B, s)</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>READ(B, t)</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>t := t + 100</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>WRITE(B, t)</strong></td>
</tr>
</tbody>
</table>
### Serializable, Not Conflict-Serializable

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_1(A) )</td>
<td>( r_2(A) )</td>
</tr>
<tr>
<td>( t := t + 100 )</td>
<td>( s := s + 200 )</td>
</tr>
<tr>
<td>( w_1(A) )</td>
<td>( w_2(A) )</td>
</tr>
<tr>
<td>( r_1(B) )</td>
<td>( r_2(B) )</td>
</tr>
<tr>
<td>( t := t + 100 )</td>
<td>( s := s + 200 )</td>
</tr>
<tr>
<td>( w_1(B) )</td>
<td>( w_2(B) )</td>
</tr>
</tbody>
</table>
Testing for Conflict-Serializability

Precedence graph:
• A node for each transaction $T_i$,
• An edge from $T_i$ to $T_j$ whenever an action in $T_i$ conflicts with, and comes before an action in $T_j$
• No edge for actions in the same transaction

• The schedule is serializable iff the precedence graph is acyclic
Example 1

\[ r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B) \]
Example 1

\[ r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B) \]
Example 1

\( r_2(A) \) \hspace{0.5cm} \( r_1(B) \)

\( r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B) \)

1 \hspace{1cm} 2 \hspace{1cm} 3
Example 1

No edge because no conflict (A != B)

\[ r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B) \]
Example 1

\[ r_2(A); w_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B) \]
Example 1

No edge because same txn (2)

r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)
Example 1

\[ r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B) \]
Example 1

$r_2(A)$  $w_1(B)$  ?

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

1  2  3
Example 1

\[ r_2(A) \quad w_3(A) \quad ? \]

\[ r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B) \]

1  2  3
Example 1

Edge! Conflict from $T_2$ to $T_3$

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

1 2 3
Example 1

Edge! Conflict from $T_2$ to $T_3$

$\text{r}_2(A); \text{r}_1(B); \text{w}_2(A); \text{r}_3(A); \text{w}_1(B); \text{w}_3(A); \text{r}_2(B); \text{w}_2(B)$
Example 1

\[ r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B) \]

And so on until compared every pair of actions…
Example 1

\begin{align*}
&\text{More edges, but repeats of the same directed edge not necessary}
\end{align*}
Example 1

This schedule is **conflict-serializable**

\[ r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B) \]
Example 2

\[ r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B) \]
Example 2

\[ r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B) \]
Example 2

This schedule is NOT conflict-serializable
View Equivalence

• A serializable schedule need not be conflict serializable, even under the “worst case update” assumption

\[ w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y); \]

Is this schedule conflict-serializable?
View Equivalence

• A serializable schedule need not be conflict serializable, even under the “worst case update” assumption

\[ w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y); \]

Is this schedule conflict-serializable? No…
View Equivalence

- A serializable schedule need not be conflict serializable, even under the “worst case update” assumption

\[ w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y); \]

Lost write

\[ w_1(X); w_1(Y); w_2(X); w_2(Y); w_3(Y); \]

Equivalent, but not conflict-equivalent
View Equivalence

 Serializable, but not conflict serializable
Two schedules $S$, $S'$ are \textit{view equivalent} if:

- If $T$ reads an initial value of $A$ in $S$, then $T$ reads the initial value of $A$ in $S'$
- If $T$ reads a value of $A$ written by $T'$ in $S$, then $T$ reads a value of $A$ written by $T'$ in $S'$
- If $T$ writes the final value of $A$ in $S$, then $T$ writes the final value of $A$ in $S'$
View-Serializability

A schedule is *view serializable* if it is view equivalent to a serial schedule

Remark:

- If a schedule is *conflict serializable*, then it is also *view serializable*
- But not vice versa
Schedules with Aborted Transactions

• When a transaction aborts, the recovery manager undoes its updates

• But some of its updates may have affected other transactions!
Schedules with Aborted Transactions

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

Abort

What’s wrong?
## Schedules with Aborted Transactions

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

Cannot abort T1 because cannot undo T2
Recoverable Schedules

A schedule is *recoverable* if:

- It is conflict-serializable, and
- Whenever a transaction T commits, all transactions that have written elements read by T have already committed.
Recoverable Schedules

A schedule is *recoverable* if:

• It is conflict-serializable, and

• Whenever a transaction T commits, all transactions that have written elements read by T have *already committed*
Recoverable Schedules

T1
R(A)
W(A)

T2
R(A)
W(A)

T1
R(A)
W(A)

T2
R(A)
W(A)

Commit

Commit

Nonrecoverable

Recoverable
Recoverable Schedules

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td>W(A)</td>
<td>R(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
<td>R(B)</td>
<td>W(B)</td>
</tr>
<tr>
<td>R(C)</td>
<td></td>
<td></td>
<td>R(C)</td>
</tr>
<tr>
<td>R(D)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Abort

How do we recover?
Cascading Aborts

• If a transaction T aborts, then we need to abort any other transaction T’ that has read an element written by T

• A schedule avoids cascading aborts if whenever a transaction reads an element, the transaction that has last written it has already committed.

We base our locking scheme on this rule!
Avoiding Cascading Aborts

Without cascading aborts

With cascading aborts

T1                  T2                  T1                  T2
R(A)                R(A)                R(A)                R(A)
W(A)                W(A)                W(A)                W(A)
R(B)                R(B)                R(B)                R(B)
W(B)                W(B)                W(B)                W(B)
...                 ...                 ...                 ...

Without cascading aborts
Review of Schedules

Serializability

- Serial
- Serializable
- Conflict serializable
- View serializable

Recoverability

- Recoverable
- Avoids cascading deletes