CS 457: Database Management Systems

Transaction Schedules
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
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
Transactions

• Major component of database systems
• Critical for most applications; arguably more so than SQL

• Turing awards to database researchers:
  – Charles Bachman 1973
  – Edgar Codd 1981 for inventing relational dbs
  – Jim Gray 1998 for inventing transactions
  – Mike Stonebreaker 2015 for INGRES and Postgres
    • And many other ideas after that
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**
Transaction Isolation
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: unrepeateable 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
A and B are elements in the database. t and s are variables in tx source code.

<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 Serial 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>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 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>
<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.
A Non-Serializable Schedule

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

T1
---
READ(A, t)
t := t+100
WRITE(A, t)

T2
---
READ(A,s)
s := s + 200
WRITE(A,s)
READ(B,s)
s := s + 200
WRITE(B,s)

READ(B, t)
t := t+100
WRITE(B,t)

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

…we don’t expect the scheduler to schedule this
Ignoring Details

• Assume worst case updates:
  – We never 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

- Conflict: pair of consecutive actions in schedule s.t. if swapped, then behavior changes
  - 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:

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

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

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

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

Example:

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

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

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

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

\[ \ldots \]

\[ r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B) \]
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 $T_j$

- The schedule is conflict-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

This schedule is conflict-serializable
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

\[
\text{Is this schedule conflict-serializable?} \quad \text{No...}
\]

\[
\text{Is this schedule serializable?} \quad \text{Yes!}
\]
View Equivalence

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

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

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

Equivalent, but not conflict-equivalent
## View Equivalence

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>W1(X)</td>
<td>W2(X)</td>
<td>CO2</td>
</tr>
<tr>
<td></td>
<td>W1(Y)</td>
<td>W2(Y)</td>
<td>CO3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Lost</strong></td>
</tr>
</tbody>
</table>

**Serializable, but not conflict serializable**
View Equivalence

Two schedules $S$, $S'$ are *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

T1

R(A)
W(A)

Abort

T2

R(A)
W(A)
R(B)
W(B)
Commit

What’s wrong?
Schedules with Aborted Transactions

What's wrong?

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 who have written elements read by T have already committed
Recoverable Schedules

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>R(A)</td>
<td>R(B)</td>
<td>R(C)</td>
</tr>
<tr>
<td>W(A)</td>
<td>W(A)</td>
<td>W(B)</td>
<td>W(C)</td>
</tr>
<tr>
<td>R(B)</td>
<td>W(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(C)</td>
<td>W(C)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abort</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
### Avoiding Cascading Aborts

<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>. . .</td>
</tr>
</tbody>
</table>

**With cascading aborts**

<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>Commit</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td>R(B)</td>
</tr>
<tr>
<td>W(B)</td>
<td>W(B)</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
</tr>
</tbody>
</table>

**Without cascading aborts**
Review of Schedules

Serializability
- Serial
- Serializable
- Conflict serializable
- View serializable

Recoverability
- Recoverable
- Avoids cascading deletes
Scheduler

• The scheduler:
  – Module that schedules the transaction’s actions, ensuring *serializability*

• Two main approaches
  • **Pessimistic**: locks
  • **Optimistic**: timestamps, multi-version, validation