# Introduction to Database Systems
# CSE 444

Lecture 13

Transactions: Concurrency Control (part 1)

# Outline

- Serial and Serializable Schedules (18.1)
- Conflict Serializability (18.2)
- Locks (18.3)

# The Problem

- Multiple transactions are running concurrently
  T1, T2, …

- They read/write some common elements
  A1, A2, …

- How can we prevent unwanted interference ?
- The SCHEDULER is responsible for that

# Some Famous Anomalies

- What could go wrong if we didn't have concurrency control:
  - Dirty reads (including inconsistent reads)
  - Unrepeatable reads
  - Lost updates

Many other things can go wrong too

# Dirty Reads

**Write-Read Conflict**

$T_1$: WRITE(A)

$T_2$: READ(A)

$T_1$: ABORT

# Inconsistent Read

**Write-Read Conflict**

$T_1$: A := 20; B := 20;
$T_1$: WRITE(A)

$T_2$: READ(A);
$T_2$: READ(B);

$T_1$: WRITE(B)

# Unrepeatable Read

**Read-Write Conflict**

$T_1$: WRITE(A)

$T_2$: READ(A);

$T_2$: READ(A);

# Lost Update

**Write-Write Conflict**

$T_1$: READ(A)

$T_1$: A := A+5

$T_1$: WRITE(A)

$T_2$: READ(A);

$T_2$: A := A*1.3

$T_2$: WRITE(A);

# Schedules

- Given multiple transactions
- A schedule is a sequence of interleaved actions from all transactions

# Example

| T1 | T2 |
|---|---|
| READ(A, t) | READ(A,s) |
| t := t+100 | s := s*2 |
| WRITE(A, t) | WRITE(A,s) |
| READ(B, t) | READ(B,s) |
| t := t+100 | s := s*2 |
| WRITE(B,t) | WRITE(B,s) |

# A Serial Schedule

| T1 | T2 | A | B |
|---|---|---|---|
| READ(A, t) | | 25 | 25 |
| t := t+100 | | | |
| WRITE(A, t) | | 125 | |
| READ(B, t) | | | |
| t := t+100 | | | |
| WRITE(B,t) | | | 125 |
| | READ(A,s) | | |
| | s := s*2 | | |
| | WRITE(A,s) | 250 | |
| | READ(B,s) | | |
| | s := s*2 | | |
| | WRITE(B,s) | | 250 |

Serial schedule: (T1,T2)

# A Serial Schedule (version 2)

| T1 | T2 | A | B |
|---|---|---|---|
| | READ(A,s) | 25 | 25 |
| | s := s*2 | | |
| | WRITE(A,s) | 50 | |
| | READ(B,s) | | |
| | s := s*2 | | |
| | WRITE(B,s) | | 50 |
| READ(A, t) | | | |
| t := t+100 | | | |
| WRITE(A, t) | | 150 | |
| READ(B, t) | | | |
| t := t+100 | | | |
| WRITE(B,t) | | | 150 |

Serial schedule: (T2,T1)

# Serializable Schedule

- A schedule is serializable if it is equivalent to a serial schedule

A schedule S is serializable, if there is a serial schedule S’, such that for every initial database state, the effects of S and S’ are the same

# A Serializable Schedule

| T1 | T2 | A | B |
|---|---|---|---|
| READ(A, t) | | 25 | 25 |
| t := t+100 | | | |
| WRITE(A, t) | | 125 | |
| | READ(A,s) | | |
| | s := s*2 | | |
| | WRITE(A,s) | 250 | |
| READ(B, t) | | | |
| t := t+100 | | | |
| WRITE(B,t) | | | 125 |
| | READ(B,s) | | |
| | s := s*2 | | |
| | WRITE(B,s) | | 250 |

Notice:
This is NOT a serial schedule

# A Non-Serializable Schedule

| T1 | T2 | A | B |
|---|---|---|---|
| READ(A, t) | | 25 | 25 |
| t := t+100 | | | |
| WRITE(A, t) | | 125 | |
| | READ(A,s) | | |
| | s := s*2 | | |
| | WRITE(A,s) | 250 | |
| | READ(B,s) | | |
| | s := s*2 | | |
| | WRITE(B,s) | | 50 |
| READ(B, t) | | | |
| t := t+100 | | | |
| WRITE(B,t) | | | 150 |

# Transaction Semantics

| T1 | T2 | A | B |
|---|---|---|---|
| READ(A, t) | | 25 | 25 |
| t := t+100 | | | |
| WRITE(A, t) | | 125 | |
| | READ(A,s) | | |
| | s := s+200 | | |
| | WRITE(A,s) | 325 | |
| | READ(B,s) | | |
| | s := s+200 | | |
| | WRITE(B,s) | | 225 |
| READ(B, t) | | | |
| t := t+100 | | | |
| WRITE(B,t) | | | 325 |

Is this serializable?

# Ignoring Details

- Serializability is undecidable!
- Scheduler should not look at transaction details
- Assume worst case updates
  - Only care about reads r(A) and writes w(A)
  - Not the actual values involved

# Notation

actions

$T_1$: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$

$T_2$: $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$

transaction

schedule

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

# Conflict Serializability

## 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

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

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

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

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

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

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

# The Precedence Graph Test

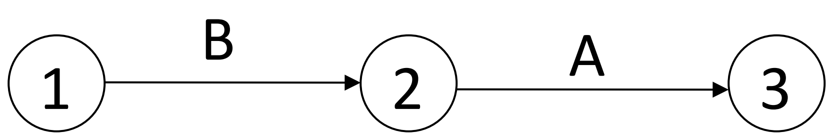
Is a schedule conflict-serializable ?

Simple test:

- Build a graph of all transactions $T_i$
- Edge from $T_i$ to $T_j$ if $T_i$ makes an action that conflicts with one of $T_j$ and comes first
- The test: if the graph has no cycles, then it is conflict serializable !

# Example 1

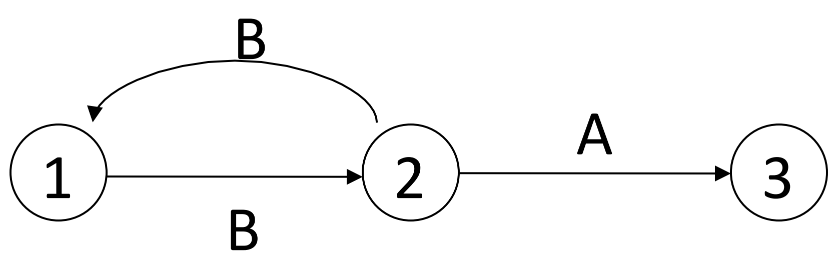
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$




This schedule is conflict-serializable

# Example 2

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




This schedule is NOT conflict-serializable

# Conflict Serializability

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

Lost write

$w_1(Y); w_2(Y); w_2(X); w_1(X); w_3(X);$

$w_1(Y); w_1(X); w_2(Y); w_2(X); w_3(X);$

Equivalent, but can’t swap

# Scheduler

- The scheduler is the module that schedules the transaction's actions, ensuring serializability
- How?  We discuss three techniques in class:
  - Locks
  - Time stamps (next lecture)
  - Validation (next lecture)

# Locking Scheduler

Simple idea:

- Each element has a unique lock
- Each transaction must first acquire the lock before reading/writing that element
- If the lock is taken by another transaction, then wait
- The transaction must release the lock(s)

# Notation

$l_i$(A) = transaction $T_i$ acquires lock for element A

$u_i$(A) = transaction $T_i$ releases lock for element A

# Example

| T1 | T2 |
|---|---|
| $L_1$(A); READ(A, t) | |
| t := t+100 | |
| WRITE(A, t); $U_1$(A); $L_1$(B) | |
| | $L_2$(A); READ(A,s) |
| | s := s*2 |
| | WRITE(A,s); $U_2$(A); |
| | $L_2$(B); **DENIED…** |
| READ(B, t) | |
| t := t+100 | |
| WRITE(B,t); $U_1$(B); | |
| | …**GRANTED;** READ(B,s) |
| | s := s*2 |
| | WRITE(B,s); $U_2$(B); |

Scheduler has ensured a conflict-serializable schedule

# Example

| T1 | T2 |
|---|---|
| $L_1$(A); READ(A, t) | |
| t := t+100 | |
| WRITE(A, t); $U_1$(A) | |
| | $L_2$(A); READ(A,s) |
| | s := s*2 |
| | WRITE(A,s); $U_2$(A); |
| | $L_2$(B); READ(B,s) |
| | s := s*2 |
| | WRITE(B,s); $U_2$(B); |
| $L_1$(B); READ(B, t) | |
| t := t+100 | |
| WRITE(B,t); $U_1$(B); | |

Locks did not enforce conflict serializability!!

# Two Phase Locking (2PL)

The 2PL rule:

- In every transaction, all lock requests must precede all unlock requests
- This ensures conflict serializability! (why?)

# Example: 2PL transactions

| T1 | T2 |
| --- | --- |
| $L_1(A)$; $L_1(B)$; READ(A, t) | |
| t := t+100 | |
| WRITE(A, t); $U_1(A)$ | |
| | $L_2(A)$; READ(A,s) |
| | s := s*2 |
| | WRITE(A,s); |
| | $L_2(B)$; **DENIED…** |
| READ(B, t) | |
| t := t+100 | |
| WRITE(B,t); $U_1(B)$; | |
| | …**GRANTED;** READ(B,s) |
| | s := s*2 |
| | WRITE(B,s); $U_2(A)$; $U_2(B)$; |

Now it is conflict-serializable

# What about Aborts?

- 2PL enforces conflict-serializable schedules
- But what if a transaction releases its locks and then aborts?
- Serializable schedule definition only considers transactions that commit
  - Relies on assumptions that aborted transactions can be undone completely

# Example with Abort

| T1 | T2 |
|---|---|
| $L_1(A)$; $L_1(B)$; READ(A, t) | |
| t := t+100 | |
| WRITE(A, t); $U_1(A)$ | |
| | $L_2(A)$; READ(A,s) |
| | s := s*2 |
| | WRITE(A,s); |
| | $L_2(B)$; **DENIED…** |
| READ(B, t) | |
| t := t+100 | |
| WRITE(B,t); $U_1(B)$; | |
| | …**GRANTED;** READ(B,s) |
| | s := s*2 |
| | WRITE(B,s); $U_2(A)$; $U_2(B)$; |
| ABORT | COMMIT |

# Strict 2PL

- Strict 2PL: All locks held by a transaction are released when the transaction is completed

- Ensures that schedules are recoverable
  - Transactions commit only after all transactions whose changes they read also commit
- Avoids cascading rollbacks

# Deadlock

- Transaction T1 waits for a lock held by T2;
- But T2 waits for a lock held by T3;
- While T3 waits for . . . .
- . . .
- . . .and T73 waits for a lock held by T1 !!

- Could be avoided, by ordering all elements (see book); or deadlock detection + rollback

# Lock Modes

- S = shared lock (for READ)
- X = exclusive lock (for WRITE)
- U = update lock
  - Initially like S
  - Later may be upgraded to X
- I = increment lock (for A := A + something)
  - Increment operations commute

Recommended reading: chapter 18.4

# The Locking Scheduler

Task 1:

Add lock/unlock requests to transactions

- Examine all READ(A) or WRITE(A) actions
- Add appropriate lock requests
- Ensure 2PL !

Recommended reading: chapter 18.5

# The Locking Scheduler

Task 2:

Execute the locks accordingly

- Lock table: a big, critical data structure in a DBMS !
- When a lock is requested, check the lock table
  - Grant, or add the transaction to the element's wait list
- When a lock is released, re-activate a transaction from its wait list
- When a transaction aborts, release all its locks
- Check for deadlocks occasionally

Recommended reading: chapter 18.5