Week 14, Lec 27

Transaction and Concurrency Control

Part 2

Contents

- **Transaction control**
- **Data Concurrency and Consistency in a Multiuser Environment**
- **Locking**

Conflict Serializability

- If a schedule *S* can be transformed into another schedule *S´* by a series of ***swaps of non-conflicting instructions***, we say that *S* and *S´* are **conflict equivalent**.
- We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule

Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

Transforming details were discussed on board in class.

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 3

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 6

Conflict Serializability (Cont.)

- Example of a schedule that is **not conflict serializable**:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

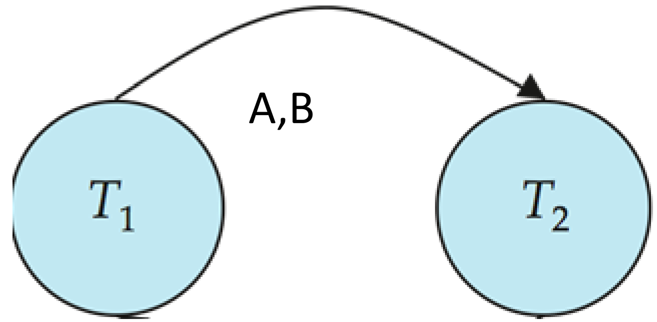
- We are **unable to swap instructions** in the above schedule **to obtain either** the **serial schedule** < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

Testing for Serializability

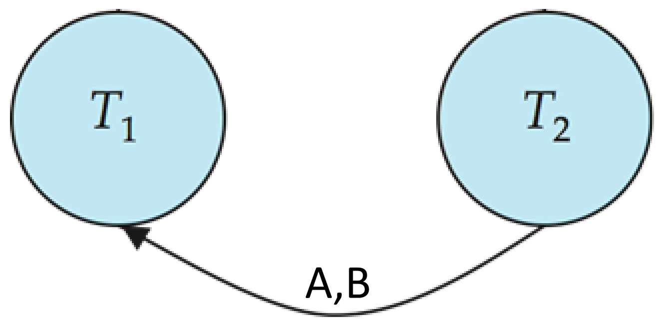
- Consider some schedule of a set of transactions $T_1, T_2, \ldots, T_n$
- **Precedence graph** — a **directed** graph where the **vertices** are the **transactions (names).**
- We draw an **arc from Ti to Tj if** the **two transaction conflict, and Ti accessed** the **data item** on which the **conflict** arose **earlier**.
- We **may label the arc by** the **item** that was **accessed**.
- **Edge: Ti -> Tj means one of 3 conditions holds**
  - Ti runs read(Q) before Tj runs write(Q)
  - Ti runs write(Q) before Tj runs read(Q)
  - Ti runs write(Q) before Tj runs write(Q)

Testing for Serializability

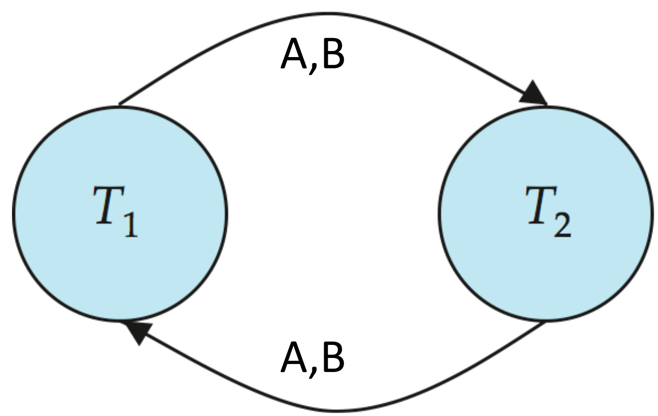
- **Example 1 – Precedence Graph for Schedule 1**



- **Example 2 – Precedence Graph for Schedule 2**



- **Example 3 – Precedence Graph for Schedule 4**

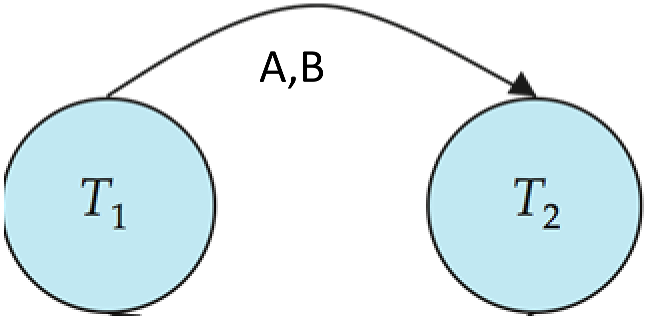


Schedule 1 and Its Precedence Graph

- A **serial** schedule in which T_1 is followed by T_2 :

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

Precedence Graph for Schedule 1

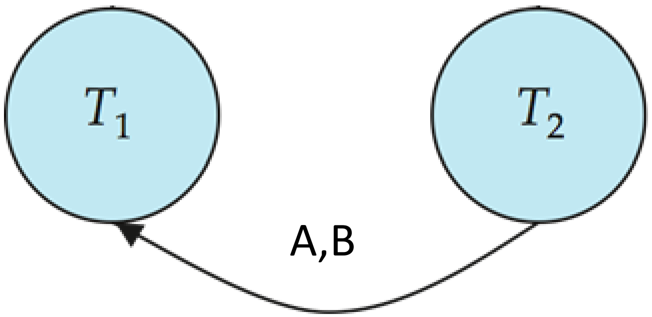


Schedule 2 and its Precedence Graph

- A **serial** schedule where T_2 is followed by T_1

T_1	T_2
	read (A) $temp$:= A * 0.1 A := A - $temp$ write (A) read (B) B := B + $temp$ write (B) commit
read (A) A := A – 50 write (A) read (B) B := B + 50 write (B) commit	

Precedence Graph for Schedule 2

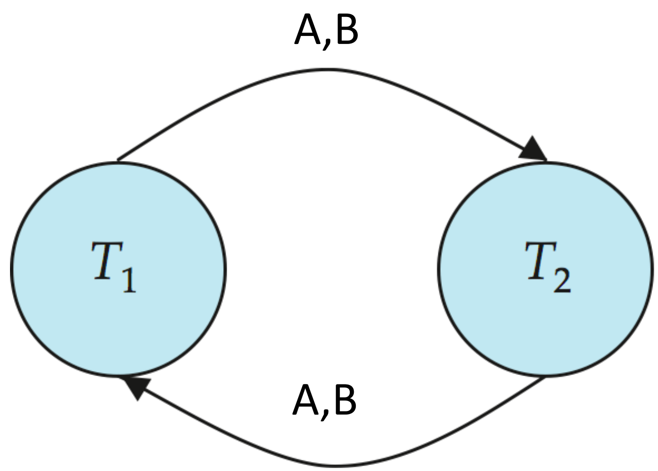


Schedule 4 and its Precedence Graph

- The following **concurrent schedule *does not* preserve** the value of (***A + B***).

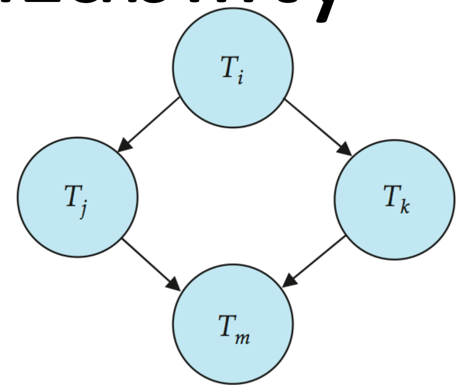
T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

Precedence Graph for Schedule 4

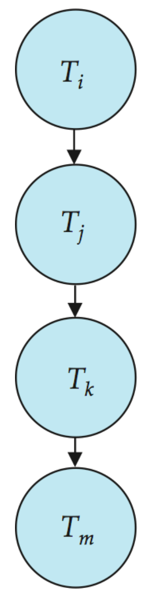


Test for Conflict Serializability

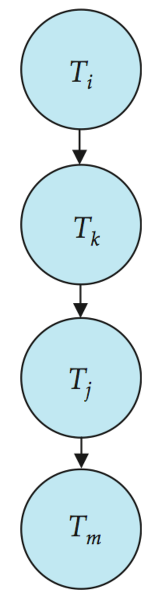
- **A schedule is conflict serializable if and only if its precedence graph is acyclic.**
- **Cycle-detection** algorithms exist which take **order *n*²** time, where *n* is the number of vertices in the graph.
- If precedence graph is acyclic, the serializability order can be obtained by a ***topological sorting*** of the graph.
  - This is a **linear order** consistent with the partial order of the graph.
  - For example, a serializability order for Schedule A would be $T_i \rightarrow T_j \rightarrow T_k \rightarrow T_m$



(a)



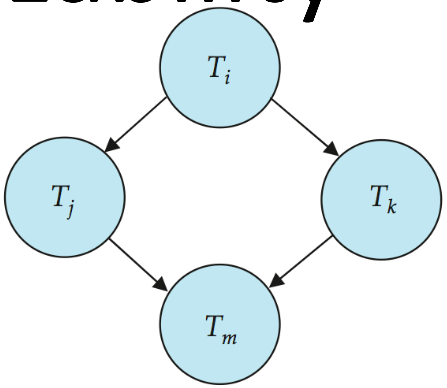
(b)



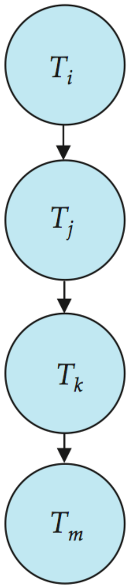
(c)

Test for Conflict Serializability

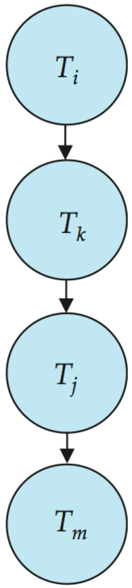
- *topological sorting* of a directed acyclic graph (DAG)
  - Ordering of vertices such that **if (u,v) is an edge**, then **u appears before v** in the **resulting order**
  - One topological sorting of Schedule (a) would be in (b) $T_i \rightarrow T_j \rightarrow T_k \rightarrow T_m$
- (c) – another acceptable serializability order of the trasactions: Ti-> Tk->Tj-> Tm.



(a)



(b)



(c)

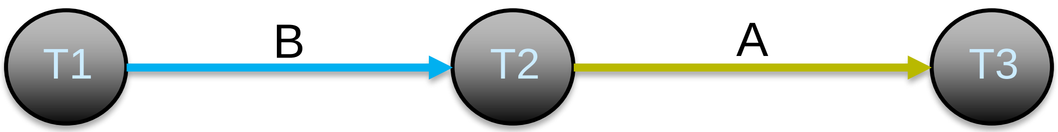
Examples: Draw Precedence Graphs

- Schedule extra 1:
  - T2:R(A),T1:R(B), T2:W(A), T3:R(A), T1:W(B), T3:W(A), T2:R(B), T2:W(B)

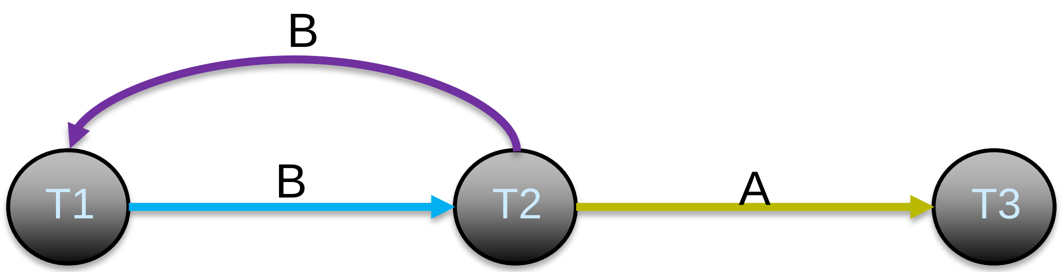
- Schedule extra 2:
  - T2:R(A),T1:R(B), T2:W(A), T2:R(B), T3:R(A), T1:W(B), T3:W(A), T2:W(B)

Examples

- Schedule extra 1:
  - T2:R(A),T1:R(B), T2:W(A), T3:R(A), T1:W(B), T3:W(A), T2:R(B), T2:W(B)



- Schedule extra 2:
  - T2:R(A),T1:R(B), T2:W(A), T2:R(B), T3:R(A), T1:W(B), T3:W(A), T2:W(B)



Data Concurrency and Consistency

- **Data concurrency**
  - Many users can access data at the same time.
- **Data consistency**
  - Each user sees a consistent view of the data
    - Including visible changes made by the user's own transactions and transactions of other users.

4 Types of Concurrency Problems that Lock can Prevent

- Lost updates
  - Two transactions select the same row, and then update the row based on the values originally selected
  - Since each transaction is unaware of the other, the later update overwrites the earlier update

4 Types of Concurrency Problems that Lock can Prevent

- Dirty reads
  - A transaction selects data that hasn’t been committed by another transaction
  - If the second transaction rolls back its changes, the first transaction selects a row that does not exist in the database.

4 Types of Concurrency Problems that Lock can Prevent

- Nonrepeatable reads
  - Two SELECT statements in one transaction that try to get the same data actually get different values because another transaction had updated the data during the time between the two statements

4 Types of Concurrency Problems that Lock can Prevent

- Phantom reads
  - Occurs when one transaction update or delete a set of rows at the same time when another transaction does insert or delete on the same set of rows
  - Example
    - TA updates payment_total for all invoices with a balance due
    - TB inserts new unpaid invoices in the mean time
    - When TA is done, invoices with balance due still exist
      - New unpaid invoices were not updated by TA

The Isolation Models

- Isolation models
  - Controls the degree to which transactions isolate from each other
    - More restrictive level will reduce or eliminate concurrency problems
    - Less restrictive level enhances performance
  - Can prevent different concurrency problems
    - Lost updates
    - Dirty reads
    - Nonrepeatable (fuzzy) reads
    - Phantom reads

Isolation Levels and Concurrency Problems

Isolation level	Problems prevented
READ UNCOMMITTED	None
READ COMMITTED	Dirty reads
REPEATABLE READ	Dirty reads, lost updates, non-repeatable read
SERIALIABLE	All

Oracle isolation levels

Read committed	Each query executed by a transaction sees only data that was committed before the **query** began (Oracle default isolation level)
Serializable	Serializable transactions see only those changes that were committed at the time the **transaction** began, plus its **own changes**
Read-only	The transaction sees only those changes that were committed at the time the transaction began and do not allow any DML statement

Set the Isolation Level

You can set the isolation level of a transaction by using one of these statements at the beginning of a transaction:

- **`SET TRANSACTION ISOLATION LEVEL READ COMMITTED;`**
- **`SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;`**
- **`SET TRANSACTION ISOLATION LEVEL READ ONLY;`**

Serializable Transaction Failure

```
SET TRANSACTION
ISOLATION
LEVEL SERIALIZABLE
```

Repeated query sees the same data, even if it was changed by another concurrent user

```
SELECT .......

SELECT .......
```

Fails if attempting to update a row changed and committed by another transaction since this transaction began

```
UPDATE .......
```

IF "Can't Serialize Access"
THEN ROLLBACK
LOOP and retry

Modes of Locking

- **Exclusive lock mode**
  - Prevents the associates resource from being shared.
  - This lock mode is obtained to modify data.
  - The first transaction to lock a resource exclusively is the only transaction that can alter the resource until the exclusive lock is released.

Modes of Locking

- Share lock mode
  - Allows the associated resource to be shared, depending on the operations involved.
  - Multiple users reading data can share the data,
    - holding share locks to prevent concurrent access by a writer (who needs an exclusive lock).
  - Several transactions can acquire share locks on the same resource.

Types of Locks

Lock	Description
DML locks (data locks)	DML locks protect data For example, table locks lock entire tables, rowlocks lock selected rows.
DDL locks (dictionary locks)	DDL locks protect the structure of schema objects
Internal locks and latches	Internal locks and latches protect internal database structures such as datafiles

Deadlock

- **Occurs when neither of two transactions can be committed**
  - **Because each transaction is waiting for the other transaction to release the lock on the resource it needs.**
  - **Example in next slide**

Deadlock

T1	TIME	T2
UPDATE emp SET sal = sal*1.1 WHERE empno = 1000;	A	UPDATE emp WHERE empno = 2000;
UPDATE emp SET sal = sal*1.1 WHERE empno = 2000;	B	UPDATE emp SET mgr = 1342 WHERE empno = 1000;
ORA-00060: deadlock detected while waiting for resource	C	

Common Recommendations

- Keep transactions as short as possible
- Avoid executing long-running queries when transactions which update the table are also executing.
- Use lowest possible transaction isolation level
- Make large changes when you are assured of nearly exclusive access