DATABASE SYSTEMS (Transaction Management)

WEEK 13 LECTURE 1& 2

AQSA IFTIKHAR

Topics to Cover

- Concurrency Control Techniques
  - 2PL
  - Time stamping
- Two Phase Locking Problems
  - Deadlocks
  - Deadlock Prevention
  - Deadlock Detection

Concurrency Control Techniques

The two main concurrency control techniques that allow transactions to execute safely in parallel subject to certain constraints are :

- Locking
- Timestamping

Locking Methods

- **Shared lock** If a transaction has a shared lock on a data item, it can read the item but not update it.
- **Exclusive lock** If a transaction has an exclusive lock on a data item, it can both read and update the item.

Locking Methods

- Locking methods are the most widely used approach to ensure serializability of concurrent transactions.
- There are several variations, but all share the same fundamental characteristic, namely that a transaction must claim a **shared** *(read)* or **exclusive** *(write)* lock on a data item before the corresponding database read or write operation.
- The **lock** prevents another transaction from modifying the item or even reading it, in the case of an exclusive lock. The basic rules for locking are as follows.

Locking Methods

- Because read operations cannot conflict, it is permissible for more than one transaction to hold shared locks simultaneously on the same item.
- On the other hand, an exclusive lock gives a transaction exclusive access to that item.
- Thus, as long as a transaction holds the exclusive lock on the item, no other transactions can read or update that data item.

Locking Methods

Locks are used in the following way:

- Any transaction that needs to access a data item must first lock the item, requesting a shared lock for read-only access or an exclusive lock for both read and write access. If the item is not already locked by another transaction, the lock will be granted.
- If the item is currently locked, the DBMS determines whether the request is compatible with the existing lock. If a shared lock is requested on an item that already has a shared lock on it, the request will be granted; otherwise, the transaction must **wait** until the existing lock is released.

Locking Methods

- A transaction continues to hold a lock until it explicitly releases it either during execution or when it terminates (aborts or commits). It is only when the exclusive lock has been released that the effects of the write operation will be made visible to other transactions.
- In addition to these rules, some systems permit a transaction to issue a shared lock on an item and then later to **upgrade** the lock to an exclusive lock.
- This in effect allows a transaction to examine the data first and then decide whether it wishes to update it. If

Two-phase locking (2PL)

- A transaction follows the two-phase locking protocol if all locking operations precede the first unlock operation in the transaction.
- According to the rules of this protocol, every transaction can be divided into two phases:
- first a **growing phase**, in which it acquires all the locks needed but cannot release any locks.
- Then a **shrinking phase**, in which it releases its locks but cannot acquire any new locks. There is no requirement that all locks be obtained simultaneously.

Two-phase locking (2PL)

- A transaction must acquire a lock on an item before operating on the item. The lock may be read or write, depending on the type of access needed.
- Once the transaction releases a lock, it can never acquire any new locks.
- If upgrading of locks is allowed, upgrading can take place only during the growing phase and may require that the transaction wait until another transaction releases a shared lock on the item.

Preventing the lost update problem using 2PL

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	write_lock(bal_x)	100
t_3	write_lock(bal_x)	read(bal_x)	100
t_4	WAIT	$bal_x = bal_x + 100$	100
t_5	WAIT	write(bal_x)	200
t_6	WAIT	commit/unlock(bal_x)	200
t_7	read(bal_x)		200
t_8	$bal_x = bal_x - 10$		200
t_9	write(bal_x)		190
t_{10}	commit/unlock(bal_x)		190

Preventing the lost update problem using 2PL

- To prevent the lost update problem occurring, T2 first requests an exclusive lock on bal_x.
- It can then proceed to read the value of bal_x from the database, increment it by £100, and write the new value back to the database.
- When T1 starts, it also requests an exclusive lock on bal_x.
- However, because the data item bal_x is currently exclusively locked by T2, the request is not immediately granted and T1 has to **wait** until the lock is released by T2.
- This occurs only once the commit of T2 has been completed.

Preventing the uncommitted dependency problem using 2PL

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		write_lock(bal_x)	100
t_3		read(bal_x)	100
t_4	begin_transaction	$bal_x = bal_x + 100$	100
t_5	write_lock(bal_x)	write(bal_x)	200
t_6	WAIT	rollback/unlock(bal_x)	100
t_7	read(bal_x)		100
t_8	$bal_x = bal_x - 10$		100
t_9	write(bal_x)		90
t_{10}	commit/unlock(bal_x)		90

Preventing the uncommitted dependency problem using 2PL

- To prevent this problem occurring, T4 first requests an exclusive lock on bal_x .
- It can then proceed to read the value of bal_x from the database, increment it by £100, and write the new value back to the database.
- When the rollback is executed, the updates of transaction T4 are undone and the value of bal_x in the database is returned to its original value of £100. When T3 starts, it also requests an exclusive lock on bal_x .
- However, because the data item bal_x is currently exclusively locked by T4, the request is not immediately granted and T3 must wait until the lock is released by T4.
- This occurs only when the rollback of T4 has been completed.

Preventing the inconsistent analysis problem using 2PL

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	write_lock(bal_x)		100	50	25	0
t_4	read(bal_x)	read_lock(bal_x)	100	50	25	0
t_5	$bal_x = bal_x - 10$	WAIT	100	50	25	0
t_6	write(bal_x)	WAIT	90	50	25	0
t_7	write_lock(bal_z)	WAIT	90	50	25	0
t_8	read(bal_z)	WAIT	90	50	25	0
t_9	$bal_z = bal_z + 10$	WAIT	90	50	25	0
t_{10}	write(bal_z)	WAIT	90	50	35	0
t_{11}	commit/unlock(bal_x, bal_z)	WAIT	90	50	35	0
t_{12}		read(bal_x)	90	50	35	0
t_{13}		sum = sum + bal_x	90	50	35	90
t_{14}		read_lock(bal_y)	90	50	35	90
t_{15}		read(bal_y)	90	50	35	90
t_{16}		sum = sum + bal_y	90	50	35	140
t_{17}		read_lock(bal_z)	90	50	35	140
t_{18}		read(bal_z)	90	50	35	140
t_{19}		sum = sum + bal_z	90	50	35	175
t_{20}		commit/unlock(bal_x, bal_y, bal_z)	90	50	35	175

Preventing the inconsistent analysis problem using 2PL

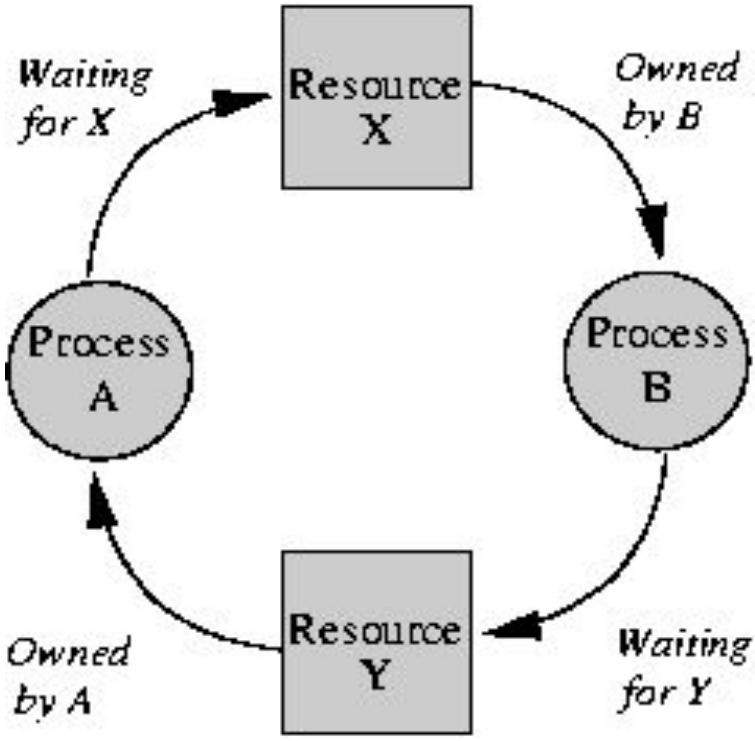
- To prevent this problem from occurring, T5 must precede its reads by exclusive locks, and T6 must precede its reads with shared locks.
- Therefore, when T5 starts, it requests and obtains an exclusive lock on bal_x. Now, when T6 tries to share lock bal_x, the request is not immediately granted and T6 has to wait until the lock is released, which is when T5 commits.

2PL Problem: Deadlock

- An situation that may result when two (or more) transactions are each waiting for locks to be released that are held by the other.

OR

- **Deadlock** is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.



Deadlock Prevention

Timeouts: A simple approach to deadlock prevention is based on ***lock timeouts.***

- In this approach, a transaction that requests a lock will wait for only a system-defined period of time.
- If the lock has not been granted within this period, the lock request times out.
- In this case, the DBMS assumes that the transaction may be deadlocked, even though it may not be, and it aborts and automatically restarts the transaction.
- This is a very simple and practical solution to deadlock prevention that is used by several commercial DBMSs.

Deadlock Prevention

Wait-Die Scheme

- In this scheme, if a transaction requests to lock a resource (data item), which is already held with a conflicting lock by another transaction, then one of the two possibilities may occur :-
- If $TS(T_i) < TS(T_j)$ − that is T_i, which is requesting a conflicting lock, is older than T_j − then T_i is allowed to wait until the data-item is available.
- If $TS(T_i) > TS(t_j)$ − that is T_i is younger than T_j − then T_i dies. T_i is restarted later with a random delay but with the same timestamp.

Deadlock Prevention

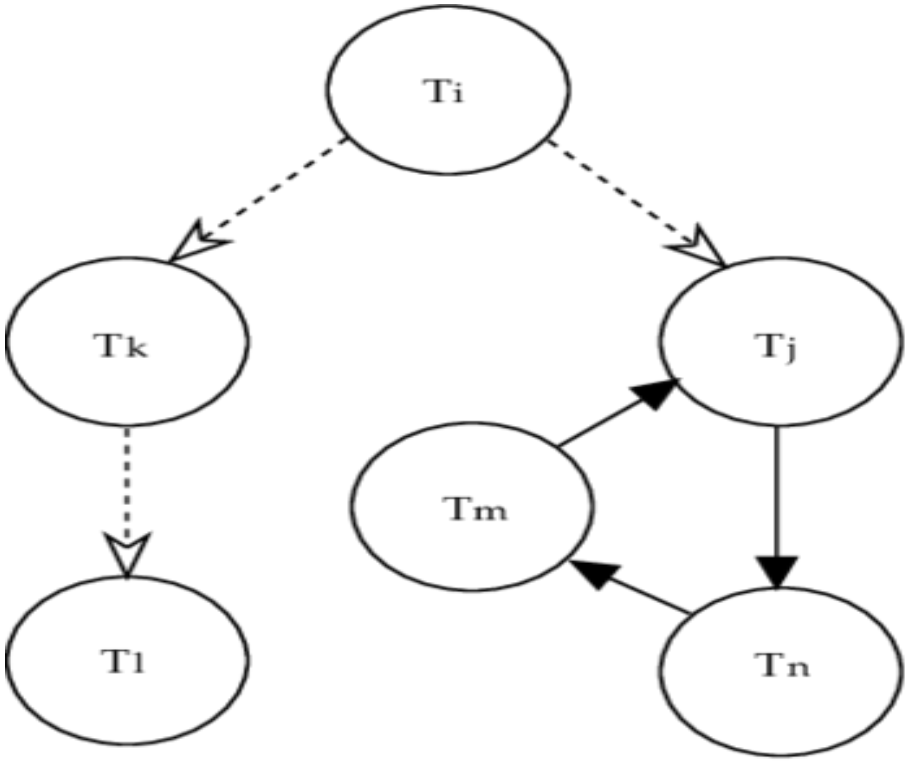
Wound-Wait Scheme

- In this scheme, if a transaction requests to lock a resource (data item), which is already held with conflicting lock by some another transaction, one of the two possibilities may occur −
- If $TS(T_i) < TS(T_j)$, then T_i forces T_j to be rolled back − that is T_i wounds T_j. T_j is restarted later with a random delay but with the same timestamp.
- If $TS(T_i) > TS(T_j)$, then T_i is forced to wait until the resource is available.

Deadlock Detection

- Deadlock detection is usually handled by the construction of a **wait-for graph** (**WFG**) that shows the transaction dependencies; that is, transaction T*i* is dependent on T*j* if transaction T*j* holds the lock on a data item that T*i* is waiting for.
- The WFG is a directed graph G 5 (N, E) that consists of a set of nodes N and a set of directed edges E, which is constructed as follows:
  - Create a node for each transaction.
  - Create a directed edge T*i* □T*j*, if transaction T*i* is waiting to lock an item that is currently locked by T*j*.
- Deadlock exists if and only if the WFG contains a cycle.

Deadlock Detection



(a) Wait-for graph with a cycle

NEXT LECTURE

- Timestamping
- How to retrieve updates due to failures
- Log
- Checkpoints