Lecture 10: Concurrency Control

CS3402 Database Systems

Database Concurrency Control

- Purposes of Concurrency Control
  - To preserve database consistency to ensure all schedules are serializable
  - To maximize the system performance (higher concurrency)
- Example: In concurrent execution environment, if T_1 conflicts with T_2 over a data item A, then the existing concurrency control decides whether T_1 or T_2 should get the A and whether the other transaction is rolled-back or waits

Two-Phase Locking Techniques

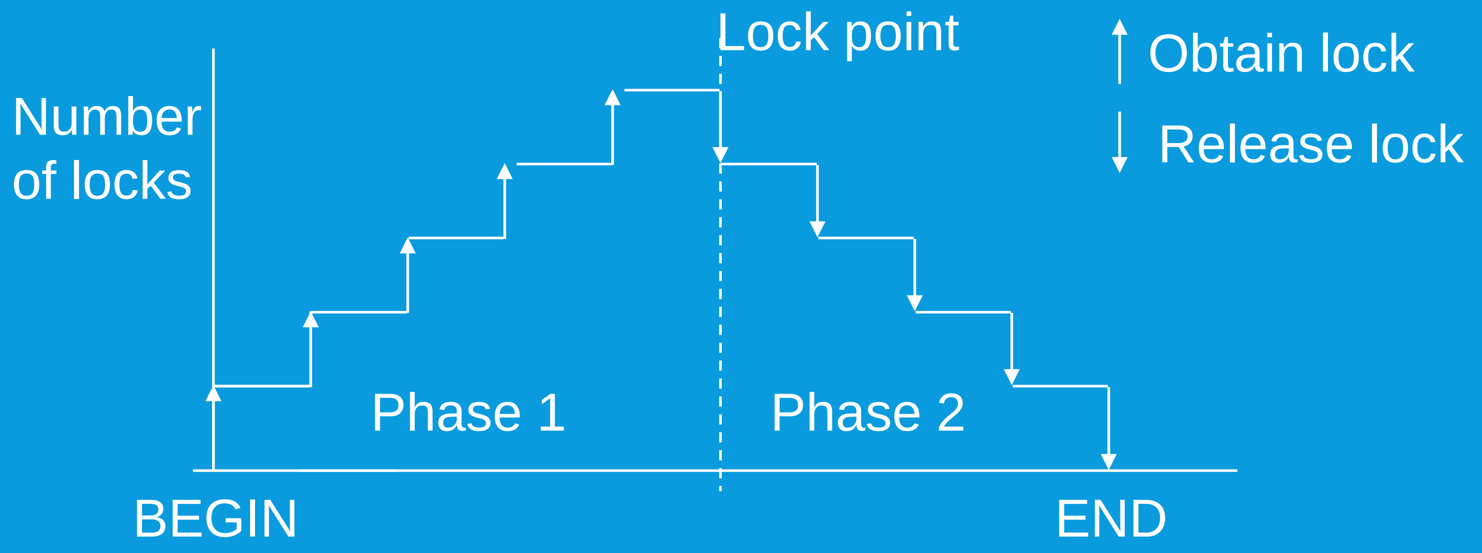
- Locking is an operation which secures
  - Permission to read a data item for a transaction
  - Permission to write (update) a data item for a transaction
  - Example: Lock(X) – data item X is locked on behalf of the requesting transaction
- Unlocking is an operation which removes these permissions from the data item
  - Example: Unlock(X) – data item X is made available to all other transactions.
- Lock and Unlock are atomic operations

Basic Two Phase Locking (B2PL) (1/2)

- Each data item has a lock associated with it, e.g., a lock entry in the lock table
- The scheduler creates a lock operation $ol_i[x]$ for each received operation $o_i[x]$
- Rules
  - When the scheduler receives an operation $p_i[x]$, it tests if $pl_i[x]$ conflicts with some $ql_j[x]$ that is already set. If so, it delays $p_i[x]$, forcing T_i to wait until it can set the lock it needs. If not, then the scheduler sets $pl_i[x]$, and sends $p_i[x]$ to the DM (data manager)
  - Once the scheduler has released a lock for a transaction, it may not subsequently obtain any more locks for that transaction (on any data item)

Basic Two Phase Locking (B2PL) (2/2)

- The two phase rule: growing phase and shrinking phase
- It can guarantee that all pairs of conflicting operations of two transactions are scheduled in the same order
  - E.g., T1 $\rightarrow$ T2 or T2 $\rightarrow$ T1 and NO T1 $\leftrightarrow$ T2



Conservative Two Phase Locking (C2PL) (1/3)

- Avoid deadlocks and abort of transactions by requiring each transaction to obtain all of its lock before any of its operations are submitted to the DM
- Each transaction pre-declares its read-set and write-set of data items to the scheduler
  - What are the locks (data items) to be accessed by the transaction?
- The scheduler tries to set all of the locks needed by the transaction ALL at ONCE
- Set lock of a transaction in one step and lock release in another step
- If all the locks can be set, the operations will be submitted to the DM for processing
- After the DM acknowledges the processing of T_i's last database operation, the scheduler may release all of T_i's locks

Conservative Two Phase Locking (C2PL) (2/3)

- If any of the locks requested in T_i's conflicts with locks presently held by other transactions in conflicting mode, the scheduler does not grant any of T_i's lock
- The scheduler inserts T_i along with its lock requests into a waiting queue
- Every time the scheduler releases the locks of a complete transaction, it examines the waiting queue to see if it can grant all the lock requests of any waiting transactions
- In Conservative 2PL, if a transaction T_i is waiting for a lock held by T_j, T_i is holding no locks (no hold and wait situation $\rightarrow$ no deadlock)

Conservative Two Phase Locking (C2PL) (3/3)

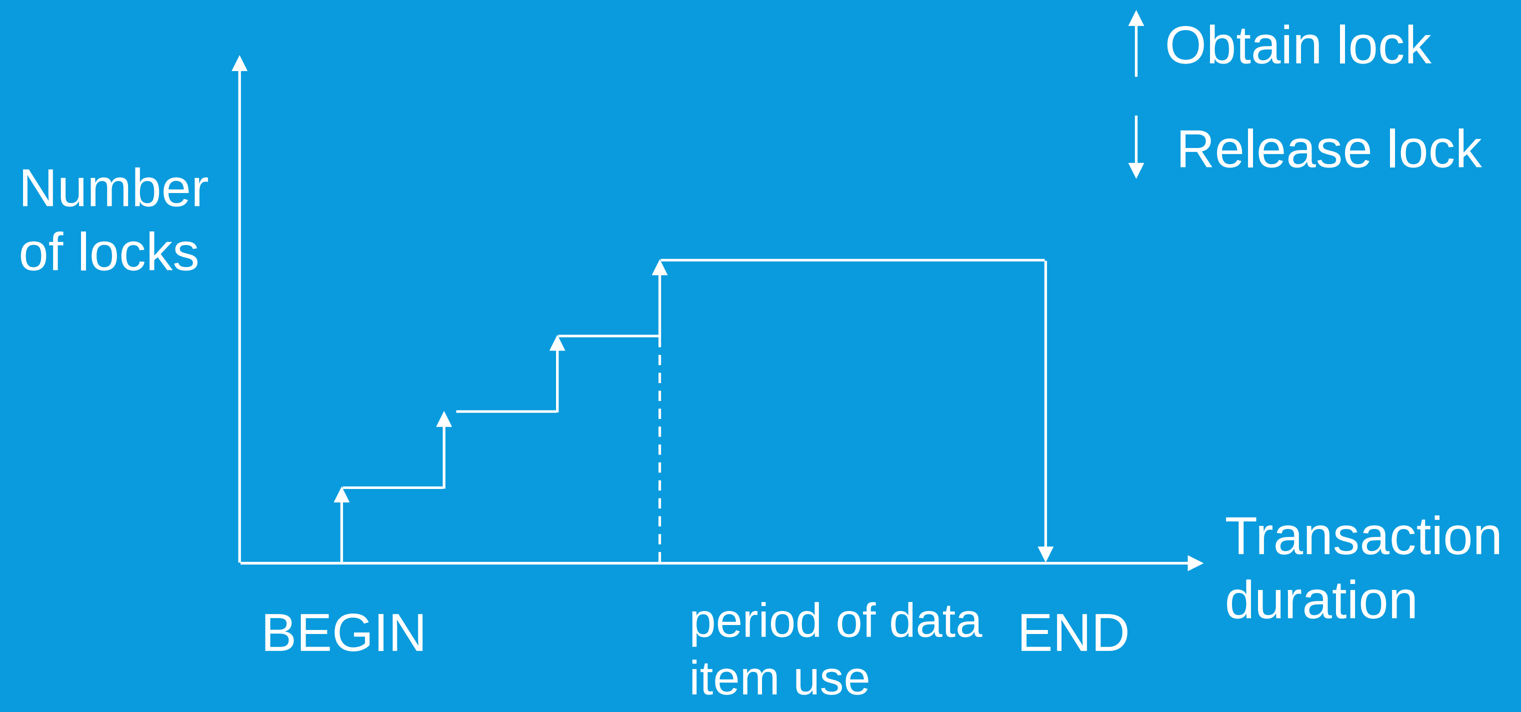
T_1	T_2
	Write(a)
Write(a)	
	Write (b)
	Commit
Write(b)	
Commit	

T_1	T_2
	WriteLock(a,b)
	Write(a)
WriteLock(a,b) $\rightarrow$ blocked	
	Write(b)
	Commit
	ReleaseLock(a,b)
WriteLock(a,b)	
Write(a)	
Write(b)	
Commit	
ReleaseLock(a,b)	

Strict Two Phase Locking (S2PL) (1/3)

- B2PL only defines the earliest time when the schedule may release a lock for a transaction
- In S2PL, it requires the scheduler to release all of a transaction's locks altogether
- T_i's locks are released after the DM acknowledge the processing of c_i (commit) or a_i (abort)
- Compared with B2PL, the lock holding time may be longer and the concurrency may be lower
- Compared with C2PL, the lock holding time may be shorter and the concurrency may be higher
- It may have the problem of deadlock but all schedules are recoverable

Strict Two Phase Locking (S2PL) (2/3)



Strict Two Phase Locking (S2PL) (3/3)

T_1	T_2
	Write(a)
Write(a)	
	Write (b)
	Commit
Write(b)	
Commit	

T_1	T_2
	WriteLock(a)
	Write(a)
WriteLock(a) → blocked	
	WriteLock(b)
	Write(b)
	Commit
	ReleaseLock(a,b)
WriteLock(a)	
Write(a)	
WriteLock(b)	
Write(b)	
Commit	
ReleaseLock(a,b)	

Performance Issues

- S2PL is better than C2PL when the transaction workload is not heavy since the lock holding time is shorter in S2PL
- When the transaction is heavy, C2PL is better than S2PL since deadlock may occur in S2PL.

Implementation Issue: Essential Components (1/2)

- Two lock modes:
  - shared (read)
  - exclusive (write).
- Shared mode: shared lock (X)
  - More than one transaction can apply shared lock on X for reading its value but no write lock can be applied on X by any other transaction.
- Exclusive mode: write lock (X)
  - Only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X.
- Conflict matrix (Permission of doing at the same time)

	Read	Write
Read	Y	N
Write	N	N

Implementation Issue: Essential Components (2/2)

- Lock Manager: Managing locks on data items
- Lock table:
  - An array to show the lock entry for each data item
  - Each entry of the array stores the identify of transaction that has set a lock on the data item including the mode
- One entry for one database? One record? One field? Lock granularity (Coarse granularity vs. fine granularity)
- Larger granularity $\rightarrow$ higher conflict probability but lower locking overhead
- How to detect lock conflict for insertion operations from a transaction?
  - A transaction reads all data items and another one inserts a new item

Implementation Issue: Lock and Unlock

- The following code performs the lock operation:

```
01: if LOCK(X) = 0 (*item is unlocked*)
02: then LOCK(X) ← 1 (*lock the item*);
03: else begin
04:     wait (until LOCK(X) = 0 and the lock
            manager wakes up the transaction);
05:     go to Line 01;
06   end;
```

- The following code performs the unlock operation:

```
01: LOCK(X) ← 0 (*unlock the
        item*);
02: if any transactions are
        waiting then wake up one of
        the waiting transactions;
```

Implementation Issue: Read Lock

- The following code performs the read lock operation:

```
01: if LOCK(X) = "unlocked" then
02:     begin
03:         LOCK(X) ← "read-locked";
04:         no_of_reads(X) ← 1;
05:     end;
06: else if LOCK(X) = "read-locked" then
07:     no_of_reads(X) ← no_of_reads(X) +1;
08: else begin
09:     wait (until LOCK(X) = "unlocked" and the lock manager wakes up the
          transaction);
10:     go to Line 01;
11: end;
```

Implementation Issue: Write Lock

- The following code performs the write lock operation:

```
01: if LOCK(X) = "unlocked" then
02:     LOCK(X) ← "write-locked";
03: else begin
04:     wait (until LOCK(X) = "unlocked" and the lock manager wakes up
           the transaction);
05:     go to Line 01
06: end;
```

Implementation Issue: Unlock

- The following code performs the unlock operation:

```
if LOCK(X) = "write-locked" then
   begin
      LOCK(X) ← "unlocked";
      wakes up one of the transactions, if any;
   end
else if LOCK(X) = "read-locked" then
   begin
      no_of_reads(X) ← no_of_reads(X) - 1;
      if no_of_reads(X) = 0 then
         begin
            LOCK(X) = "unlocked";
            wake up one of the transactions, if any;
         end;
   end;
```

Implementation Issue: Lock Conversion

- Lock conversion (read lock to write lock)
  - Lock upgrade: existing read lock to write lock

```
if T_i has a read-lock(X) and T_j has no read-lock(X) (i ≠ j) then
    convert read-lock(X) to write-lock(X);
else
    force T_i to wait until T_j unlocks X;
```

  - Lock down grade: existing write lock to read lock

```
if T_i has a write-lock(X) (*no transaction can have any lock on X*)
    convert write-lock(X) to read-lock(X);
```

Deadlock

- T1 and T2 do follow two-phase policy but they are deadlock

T_1	T_2
Read_lock(Y);	
Read_item(Y);	
	Read_lock(X);
	Read_item(X);
Write_lock(X); (waits for X)	
	Write_lock(Y); (waits for Y)

Deadlock Prevention

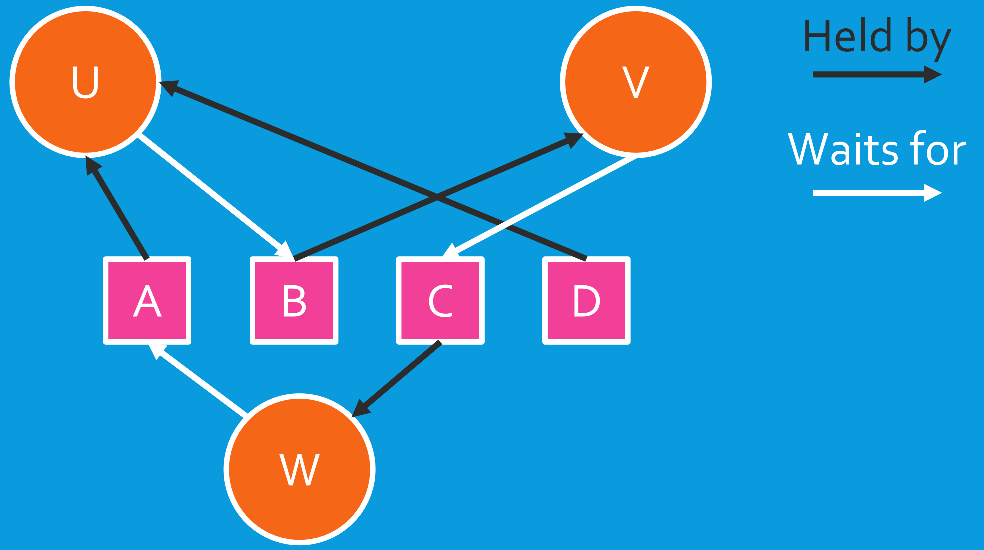
- Deadlock condition
  - Hold and wait
  - Cyclic wait
- Deadlock Prevention
  - A transaction locks all data items it refers to before it begins execution
  - This way of locking prevents deadlock since a transaction never waits for a data item
  - The conservative two-phase locking uses this approach
  - Not hold and wait condition

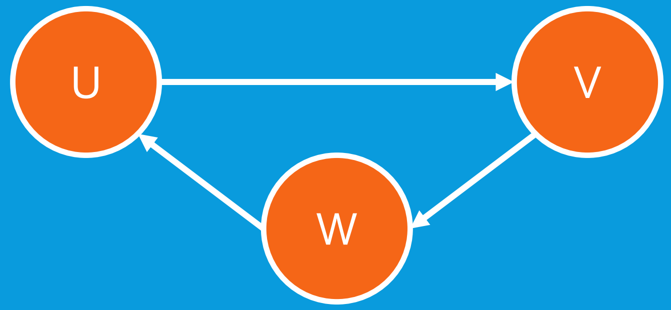
Deadlock Detection and Resolution

- Detection: In some approaches, deadlocks are allowed to happen e.g., in Strict 2PL. The scheduler maintains a wait-for-graph for detecting cycle. If a cycle exists, then one transaction involved in the cycle is selected (victim) and rolled-back.
- Resolution: A wait-for-graph is created using the lock table. As soon as a transaction is blocked, it is added to the graph. When a chain like: T_i waits for T_j, T_j waits for T_k and T_k waits for T_i occurs, this creates a cycle. One of the transactions will be chosen to abort.

Deadlock Example

U		V		W	
Write(D)	Lock D				
		Write(B)	Lock B		
Write(A)	Lock A				
				Write(C)	Lock C
Write(B)	Blocked				
		Write(C)	Blocked		
				Write(A)	Blocked





wait-for-graph

Starvation

- Starvation occurs when a particular transaction consistently waits or restarts and never gets a chance to proceed further
- In a deadlock resolution it is possible that the same transaction may consistently be selected as victim and rolled back

Deadlock Prevention Using Timestamp (1/2)

- Deadlock prevention: prevent potential deadlock to become deadlock
- Each transaction is assigned a unique time-stamp, e.g., its creation time (distributed databases: creation time + site ID)
- Wait-die Rule (non-preemptive):
  - If T_i requests a lock that is already locked by T_j, T_i is permitted to wait if and only if T_i is older than T_j (T_i's timestamp is smaller than that of T_j)
  - If T_i is younger than T_j, T_i is restarted with the same timestamp
  - When T_i requests access to the same lock in the second time, T_j may already have finished its execution

Deadlock Prevention Using Timestamp (2/2)

- Wound-Wait Rule (preemptive*):
  - If T_i requests a lock that is already locked by T_j, T_i is permitted to wait if and only if T_i is younger than T_j
  - Otherwise, T_j is restarted (with the same timestamp) and the lock is granted to T_i

(*The executing process in preemptive scheduling is interrupted in the middle of execution when higher priority one comes)

Deadlock Avoidance Using Timestamp

- If $TS(T_i) < TS(T_j)$, T_i waits else T_i dies (Wait-die)
- If $TS(T_i) < TS(T_j)$, T_j wounds else T_i waits (Wound-wait)
- Note a smaller TS means the transaction is older
- Note both methods restart the younger transaction
- Both methods prevent cyclic wait:
  - Consider this deadlock cycle: $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$
  - It is impossible since if $T_1 \ldots \rightarrow T_n$, then T_n is not allowed to wait for T_1
  - Wait-die: Older transaction is allowed to wait
  - Wound-wait: Older transaction is allowed to get the lock

Deadlock Example

- TS of T_1 < TS of T_2

T_1	T_2
Read(A);	
Write(B);	
	Read(C);
	Write(A); (blocked)
Write(C); (blocked and deadlock formed)	

Deadlock Example (wait-die)

- TS of T_1 < TS of T_2

T_1	T_2
ReadLock(A); Read(A);	
WriteLock(B); Write(B);	
	ReadLock(C); Read(C);
	Write(A); (restarts because it is younger than T_1 and T_2 releases its read lock on C before it restarts)
WriteLock(C); Write(C);	
ReleaseLock(T_1);	
	ReadLock(C); Read(C);

Deadlock Example (wound-wait)

- TS of T_1 < TS of T_2

T_1	T_2
ReadLock(A); Read(A);	
WriteLock(B); Write(B);	
	ReadLock(C); Read(C);
	WriteLock(A); (blocked because T_2 is younger than T_1)
WriteLock(C); Write(C); (T_2 is restarted by T_1 because T_2 is younger than T_1. The write lock on C is granted to T_1 after T_2 has released its read lock on C)	
ReleaseLock(T_1);	
	ReadLock(C); Read(C);
	WriteLock(A); Write(A);