Lecture 10

Concurrency Control

Dr. Vandana Kushwaha

Department of Computer Science

Institute of Science, BHU, Varanasi

Introduction

- One of the **fundamental properties** of a **transaction** is **Isolation.**
- When several **transactions** execute **concurrently** in the database, however, the **isolation property** may **no longer** be **preserved.**
- To ensure that it is, the system must **control the interaction** among the **concurrent transactions**.
- This **control** is achieved through one of a variety of **mechanisms** called **Concurrency-control schemes.**
- There are **two** main categories of **Concurrency-control schemes**:
  - **Lock-Based Protocols**
  - **Time-stamp Based Protocols**

Lock-Based Protocols

- One way to **ensure serializability** is to require that data items be accessed in a **mutually exclusive manner.**
- That is, while **one transaction is accessing** a **data item**, no other **transaction** can **modify** that data item.
- The most **common method** used to implement this requirement is to allow a transaction to access a **data item** only if it is currently **holding a lock on that item**.
- There are **two modes** in which a **data item** may be **locked:**
  - **Shared Mode**
  - **Exclusive Mode**

Lock-Based Protocols

1. **Shared Mode**

- If a **transaction Ti** has obtained a **shared-mode lock** (denoted by **S**) on item **Q**, then **Ti can read, but cannot write, Q.**

2. **Exclusive Mode**

- If a **transaction Ti** has obtained an **exclusive-mode lock** (denoted by **X**) on item **Q**, then **Ti can both read and write Q.**
- Every **transaction request** a **lock** to **Concurrency-control manager** in an **appropriate mode** on **data item Q**, depending on the types of **operations** that it will perform on Q.
- The **transaction can proceed** with the operation only after the **Concurrency-control manager grants** the **lock** to the **transaction.**

Lock-Based Protocols

- **Lock-compatibility matrix**
- The **compatibility** relation **between** the **two modes of locking** can be represented using a **matrix .**
- An **element comp(A, B)** of the **matrix** has the **value true** if and only if **mode A** is **compatible** with **mode B.**

	S	X
S	true	false
X	false	false

- Note that **shared mode is compatible** with **shared mode,** but **not** with **exclusive mode.**
- At any time, **several shared-mode locks** can be held **simultaneously** (by different transactions) on a **particular data item**.
- A subsequent **exclusive-mode lock** request has to **wait until the currently held shared-mode locks are released.**

Lock-Based Protocols

- A **transaction** requests a **shared lock** on **data item Q** by executing the **lock-S(Q) instruction.**
- Similarly, a **transaction requests** an **exclusive lock** through the **lock-X(Q) instruction.**
- A **transaction can unlock** a **data item Q** by the **unlock(Q) instruction.**
- To **access a data item**, **transaction Ti** must **first lock that item**.
- If the **data item is already locked** by **another transaction** in an **incompatible mode**, the **concurrency control manager** will **not grant the lock** until all **incompatible locks** held by other **transactions** have been **released.**
- Thus**, Ti is made to wait** until **all incompatible locks** held by other transactions have been **released.**

Example

T_1: lock-X(B);
read(B);
$B := B - 50$;
write(B);
unlock(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(A).

T_2: lock-S(A);
read(A);
unlock(A);
lock-S(B);
read(B);
unlock(B);
display($A + B$).

T_1	T_2	concurrency-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_2)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

- In this case, **transaction T2** displays **incorrect value.**
- The **reason** for this mistake is that the **transaction T1 unlocked data item B too early,** as a result of which **T2 saw** an **inconsistent state.**

Example

T_3: lock-X(B);
read(B);
$B := B - 50$;
write(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(B);
unlock(A).

T_4: lock-S(A);
read(A);
lock-S(B);
read(B);
display($A + B$);
unlock(A);
unlock(B).

- **Schedule** corresponding to **T3 and T4** will **not lead to inconsistency.**

Lock-Based Protocols

- Unfortunately, **locking can lead** to an **undesirable situation.**
- Consider the **partial schedule** of Figure for **T3** and **T4**.

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

- Since **T3 is holding an exclusive-mode lock on B** and **T4 is requesting a shared-mode lock on B, T4 is waiting for T3 to unlock B.**
- Similarly, **since T4 is holding a shared-mode lock on A** and **T3 is requesting an exclusive-mode lock on A, T3 is waiting for T4 to unlock A.**
- Thus, we have arrived at a **state where neither of these transactions can ever proceed with its normal execution.**
- This **situation** is called **Deadlock.**

Lock-Based Protocols

- When **deadlock occurs**, the **system must roll back** one of the two **transactions.**
- Once a **transaction has been rolled back**, the **data items** that were **locked by that transaction are unlocked.**
- These **data items** are **then available to the other transaction**, which can continue with its execution.
- Thus it require that **each transaction** in the system **follow a set of rules**, called a **locking protocol**, indicating when a transaction may lock and unlock each of the data items.

Starvation

- **T2**- **lock-S(x)** **Granted**
- **T1**- **lock-X(x)** **Wait for T2**
- **T3**- **lock-S(x)** **Granted**
- **Commit T2** **unlock(x)**
- **T1 still wait** for **T3 to unlock data item x.**
- The **transaction T1** may **never make progress**, and is said to be **starved.**
- We can **avoid starvation** of transactions by **granting locks** in the **following manner**:
- When a transaction **Ti requests a lock** on a **data item Q** in a particular **mode M,** the **Concurrency-control manager grants the lock** provided that :

1. There is **no other other transaction** holding a lock on **Q** in a **conflicting mode**.

2. There is **no other transaction** that is **waiting for a lock on Q**, and that **made its lock request before Ti.**

The Two-Phase Locking Protocol

- One **protocol** that **ensures serializability** is the **two-phase locking protocol.**
- This **protocol requires that** each **transaction issue lock and unlock requests in two phases:**
  - 1. **Growing phase.** A transaction may **obtain locks**, but may **not release** any lock.
  - 2. **Shrinking phase.** A transaction **may release locks**, but **may not obtain** any **new locks**.
- Initially, a **transaction is in the growing phase**.
- The **transaction acquires locks** as needed.
- Once the **transaction releases a lock**, it **enters the shrinking phase**, and it can issue no more lock requests.

The Two-Phase Locking Protocol

- **Example**: **transactions T3 and T4 are two phase.**

```
T3: lock-X(B);
    read(B);
    B := B − 50;
    write(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(B);
    unlock(A).
```

```
T4: lock-S(A);
    read(A);
    lock-S(B);
    read(B);
    display(A + B);
    unlock(A);
    unlock(B).
```

- On the other hand, transactions **T1 and T2 are not two phase.**

```
T1: lock-X(B);
    read(B);
    B := B − 50;
    write(B);
    unlock(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(A).
```

```
T2: lock-S(A);
    read(A);
    unlock(A);
    lock-S(B);
    read(B);
    unlock(B);
    display(A + B).
```

The Two-Phase Locking Protocol

- Note that the **unlock instructions do not need to appear** at the **end of the transaction.**
- **Example:** in the case of **transaction T3**, we could **move the unlock(B)** instruction to **just after the lock-X(A) instruction**, and **still retain** the **two-phase locking property.**

T_3: lock-X(B);
read(B);
$B := B - 50$;
write(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(B);
unlock(A).

- **Note:** the **two-phase locking protocol** ensures **Conflict serializability**.

The Two-Phase Locking Protocol

- **Cascading rollback** may occur under **two-phase locking**.
- Consider the **partial schedule** of Figure.

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

- Each **transaction** observes the **two-phase locking protocol**, but the **failure of T5 after the read(A) step** of **T7** leads to **cascading rollback** of **T6** and **T7.**

Strict two-phase locking protocol

- **Cascading rollbacks** can be **avoided** by a **modification of two-phase locking** called the **Strict two-phase locking protocol.**
- This **protocol** requires not only that locking be two phase, but also that **all exclusive-mode locks taken** by a **transaction be held until that transaction commits.**
- This requirement ensures that **any data written by an uncommitted transaction are locked in exclusive mode until the transaction commits**, preventing any other transaction from reading the data.
- **Another variant** of **two-phase locking** is the **Rigorous two-phase locking protocol**, which requires that **all locks be held until the transaction commits**.

Lock Conversion

T_8: read(a_1);
read(a_2);
. . .
read(a_n);
write(a_1).

T_9: read(a_1);
read(a_2);
display(a_1 + a_2).

- If we employ the **two-phase locking protocol**, then **T8 must lock a1 in exclusive mode.**
- Therefore, any **concurrent execution** of **both transactions** amounts to a **serial execution.**
- Notice, however, that **T8 needs an exclusive lock on a1** only **at the end of its execution, when it writes a1.**
- Thus, **if T8 could initially lock a1 in shared mode**, and then **could later change the lock to exclusive mode**, we **could get more concurrency**, since **T8 and T9 could access a1 and a2 simultaneously.**

Lock Conversion

- This observation leads us to a **refinement of the basic two-phase locking protocol**, in which **lock conversions** are allowed.
- There is a **mechanism for upgrading a shared lock** to an **exclusive lock**, and **downgrading** an **exclusive lock to a shared lock.**
- Conversion from **shared to exclusive modes** denoted by **upgrade**, and from **exclusive to shared** by **downgrade.**
- **Lock conversion** cannot be allowed **arbitrarily.**
- Rather, **upgrading can take place** in only the **growing phase**, whereas **downgrading can take place** in only the **shrinking phase**.

T_8	T_9
lock-S(a_1)	
	lock-S(a_1)
lock-S(a_2)	
	lock-S(a_2)
lock-S(a_3)	
lock-S(a_4)	
	unlock(a_1)
	unlock(a_2)
lock-S(a_n)	
upgrade(a_1)	

Timestamp-Based Protocols for Concurrency Control

Timestamp-Based Protocols

- **Timestamps**
- With each **transaction Ti** in the system, we associate a **unique fixed timestamp**, denoted by **TS(Ti).**
- This **timestamp** is **assigned by** the **database system** before the transaction **Ti** starts execution.
- If a **transaction Ti** has been assigned **timestamp TS(Ti)**, and a new **transaction Tj** enters the system, then **TS(Ti) < TS(Tj).**
- There are **two** simple **methods** for implementing the **timestamp** scheme:
  - **System clock**
  - **Logical counter.**

– The **timestamps** of the **transactions determine** the **serializability order.**

Timestamp-Based Protocols

- Thus, **if TS(Ti) < TS(Tj),** then the **system** must **ensure** that the produced **schedule** is **equivalent** to a **serial schedule** in which **transaction Ti appears before transaction Tj .**
- To **implement** this scheme, we associate with each **data item Q two timestamp values:**
  - **W-timestamp(Q)** denotes the **largest timestamp** of any **transaction** that **executed write(Q)** successfully.
  - **R-timestamp(Q)** denotes the **largest timestamp** of any **transaction** that **executed read(Q)** successfully.
- These **timestamps** are **updated** whenever a new **read(Q)** or **write(Q)** instruction is executed.

The Timestamp-Ordering Protocol

- The **Timestamp-ordering protocol** ensures that **any conflicting read** and **write operations** are **executed** in **timestamp order.**
- This **protocol** operates as follows:
- **1.** Suppose that **transaction Ti** issues **read(Q).**
  - **a.** If **TS(Ti) < W-timestamp(Q),** then
    - **Ti needs** to **read a value of Q** that was **already overwritten**.
    - Hence, the **read operation** is **rejected,** and **Ti is rolled back.**
  - **b.** If **TS(Ti) ≥ W-timestamp(Q)**, then
    - The **read operation** is **executed,** and
    - **R-timestamp(Q)**)= **MAX(R-timestamp(Q) , TS(Ti)).**

The Timestamp-Ordering Protocol

2. Suppose that **transaction Ti** issues **write(Q).**

- **a.** If **TS(Ti) < R-timestamp(Q),** then
  - the **value of Q** that **Ti** is **producing** was **needed previously**, and the **system assumed** that that **value would never** be **produced**.
  - Hence, the **system rejects** the **write operation** and **rolls Ti back.**
- **b.** If **TS(Ti) < W-timestamp(Q)**, then
  - **Ti** is **attempting to write** an **obsolete value** of **Q.**
  - Hence, the **system rejects** this **write operation** and **rolls Ti back**.
- **c.** Otherwise, the **system executes** the **write operation** and **sets W-timestamp(Q) to TS(Ti).**
- If a **transaction Ti** is **rolled back** by the **concurrency-control scheme** the **system assigns** it a **new timestamp** and **restarts it.**

The Timestamp-Ordering Protocol

T_{14}: read(B);
read(A);
display($A + B$).

T_{15}: read(B);
$B := B - 50$;
write(B);
read(A);
$A := A + 50$;
write(A);
display($A + B$).

T_{14}	T_{15}
read (B)	
	read (B)
	$B := B - 50$
	write (B)
read (A)	
	read (A)
display ($A + B$)	
	$A := A + 50$
	write (A)
	display ($A + B$)

- In **timestamp protocol**, a **transaction** is assigned a **timestamp** immediately **before** its **first instruction**.
- Thus, in **schedule** of Figure , **TS(T14) < TS(T15).**
- The **Timestamp-ordering protocol** ensures **Conflict Serializability.**
- This is because **conflicting operations** are **processed** in **timestamp order.**
- The **protocol ensures freedom from deadlock**, since **no transaction ever waits**.
- However, **there is a possibility** of **starvation of long transactions** if a **sequence of conflicting short transactions** causes **repeated restarting** of the **long transaction**.

Thomas’ Write Rule

- **Thomas’ Write Rule** is a **modification** to the **timestamp-ordering protocol** that allows **greater potential concurrency**.
- Let us consider **schedule** of Figure and **apply** the **timestamp-ordering protocol**.

T_{16}	T_{17}
read(Q)	
	write(Q)
write(Q)	

- Since **T16 starts before T17**, we shall assume that **TS(T16) < TS(T17).**
- **The read(Q) operation of T16** succeeds, as does the **write(Q) operation of T17.**
- When **T16 attempts its write(Q) operation**, we find that **TS(T16) < W-timestamp(Q)**, since **W-timestamp(Q) = TS(T17).**
- Thus, the **write(Q) by T16** is **rejected** and **transaction T16** must be **rolled back.**
- Any **transaction Ti** with **TS(Ti) < TS(T17)** that **attempts a read(Q)** will be **rolled back,** since **TS(Ti) < W-timestamp(Q).**

Thomas' Write Rule

- Any **transaction Tj** with **TS(Tj) > TS(T17)** must **read the value of Q written by T17,** rather than the value written by **T16.**
- The **obsolete write operations can be ignored.**
- **Thomas' write rule**:
- Suppose that transaction **Ti** issues **write(Q).**
- **1.** If **TS(Ti) < R-timestamp(Q),** then the **value of Q** that **Ti** is producing was **previously needed,** and it had been assumed that the value would never be produced.
  - Hence, the **system rejects the write operation** and **rolls Ti back.**
- **2.** If **TS(Ti) < W-timestamp(Q),** then Ti is **attempting to write** an **obsolete value of Q**.
  - Hence, this **write operation can be ignored**.
- **3. Otherwise,** the system **executes the write operation** and sets **W-timestamp(Q) to TS(Ti).**

Deadlock Handling

- A **system** is in a **deadlock state** if there exists a **set of transactions** such that **every transaction** in the **set** is **waiting for another transaction** in the **set.**
- More precisely, there exists a **set of waiting transactions {T0, T1, . . ., Tn}** such that
  - **T0** is waiting for a data item that **T1 holds**, and
  - **T1** is waiting for a data item that **T2 holds**, and ...,
  - .. and **Tn−1** is waiting for a data item that **Tn holds**, and
  - **Tn** is waiting for a data item that **T0 holds.**
- **None of the transactions** can **make progress** in such a situation.
- The **only remedy** to this undesirable situation is for the system to **rolling back some** of the **transactions** involved in the **deadlock.**
- **Rollback** of a **transaction** may be **partial:** That is, a transaction may be rolled back to the point where it obtained a lock whose release resolves the **deadlock.**

Deadlock Handling

- There are **two principal methods** for dealing with the **deadlock problem:**
- **Deadlock Prevention**
  - Deadlock prevention protocol to ensure that the system will never enter a deadlock state.
- **Deadlock Detection and Recovery**
  - We can allow the system to enter a deadlock state, and
  - then try to recover by using a deadlock detection and deadlock recovery scheme.

Deadlock Prevention

- The simplest scheme of deadlock prevention requires that **each transaction locks all its data items** before it **begins execution.**
- Moreover, **either all are locked** in **one step** or **none are locked.**
- There are two main **disadvantages** to this protocol:
  - **(1)** it is often **hard to predict**, before the transaction begins, what data items need to be locked.
  - **(2) data-item utilization** may be **very low**, since many of the data items may be locked but unused for a long time.

Deadlock Prevention

- Two different **deadlock prevention schemes** using **timestamps** have been proposed:
- **1. Wait–die scheme (**a **non-preemptive technique)**
  - When **transaction Ti** requests a **data item** currently **held by Tj** ,
  - **Ti** is allowed to **wait** only **if TS(Ti)<TS(Tj)** (that is, Ti is older than Tj).
  - Otherwise, **Ti** is **rolled back** (dies).
- **Example**
- Suppose that transactions T22, T23, and T24 have timestamps 5, 10, and 15, respectively.
- If T22 requests a data item held by T23, then **T22 will wait**.
- If T24 requests a data item held by T23, then **T24** will be **rolled back.**

Deadlock Prevention

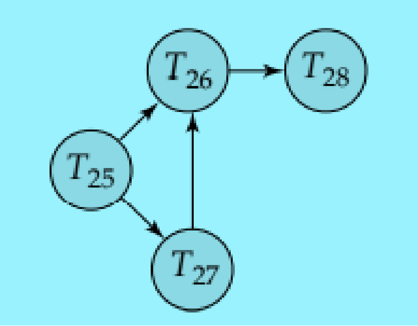
- **Wound–wait scheme(preemptive technique)**
  - When **transaction Ti** requests a **data item** currently held by **Tj** ,
  - **Ti is allowed** to **wait** only if **TS(Ti) > TS(Tj)** (that is, **Ti is younger than Tj**).
  - **Otherwise**, **Tj is rolled back** (Tj is wounded by Ti).
- **Example**
- With transactions **T22, T23,** and **T24,** if T22 requests a data item held by T23, then the data item will be **preempted** from T23, and T23 will be rolled back.
- If T24 requests a data item held by T23, then T24 will wait.
- Whenever the system rolls back transactions, it is important to ensure that there is **no starvation**—that is, no transaction gets rolled back repeatedly and is never allowed to make progress.
- Both the **wound–wait** and the **wait–die** schemes **avoid starvation.**

Deadlock Detection

- **Deadlocks** can be described precisely in terms of a **directed graph** called a **wait-for graph.**
- This **graph** consists of a **pair G = (V, E),** where V is a set of vertices and E is a set of edges.
- The **set of vertices** consists of **all the transactions** in the system.
- Each element in the set **E of edges** is an ordered pair **$T_i \rightarrow T_j$.**
- When **transaction T_i requests a data item currently being held by transaction T_j ,** then the **edge $T_i \rightarrow T_j$** is inserted in the **wait-for graph.**
- **This edge** is **removed** only when transaction T_j is no longer holding a data item needed by transaction T_i.
- A **deadlock exists** in the system if and only if the **wait-for graph contains a cycle.**
- Each transaction involved in the cycle is said to be **deadlocked**.

Deadlock Detection

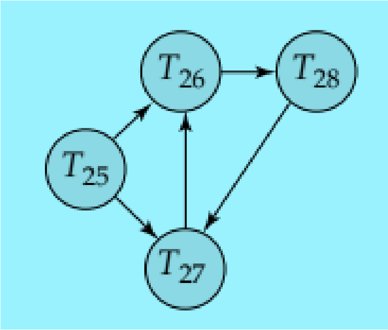
- **Example:**
- Consider the **wait-for graph** in Figure which depicts the following situation:



- Transaction T25 is waiting for transactions T26 and T27.
- Transaction T27 is waiting for transaction T26.
- Transaction T26 is waiting for transaction T28.
- Since the **graph has no cycle**, the **system is not** in a **deadlock state.**

Deadlock Detection

- Suppose now that **transaction T28** is requesting an item held by **T27.**
- The **edge T28 → T27** is added to the **wait-for graph**, resulting in the new system state in Figure.



- This time, the **graph contains the cycle** T26 → T28 → T27 → T26 implying that **transactions T26, T27, and T28** are all **deadlocked.**

Recovery from Deadlock

- **The most common solution for Deadlock Recovery** is to **roll back** one or more **transactions** to **break** the **deadlock**.
- **Three actions** need to be taken:
- **1. Selection of a victim.**
- Given a set of deadlocked transactions, we must **determine which transaction (or transactions) to roll back** to break the deadlock.
- We should **roll back those transactions** that will incur the **minimum cost.**
- **Many factors** may determine the **cost of a rollback**, including
  - **a.** How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
  - **b.** How many data items the transaction has used.
  - **c.** How many more data items the transaction needs for it to complete.
  - **d.** How many transactions will be involved in the rollback.

Recovery from Deadlock

2. Rollback

- Once we have decided that a particular transaction must be rolled back, we must determine **how far this transaction should be rolled back**.
- The **simplest solution** is a **total rollback**: Abort the transaction and then restart it.
- However, it is **more effective to roll back the transaction only as far as necessary to break the deadlock.**

3. Starvation

- In a system where the selection of victims is based primarily on cost factors, it **may happen that the same transaction is always picked as a victim.**
- As a result, this **transaction never completes** its designated task, thus **there is starvation.**
- We must ensure that **transaction can be picked as a victim only a (small) finite number of times.**