**UNIT IV- Concurrency Control**

When several transactions execute concurrently in the database, the system must control the interaction among the transactions through several mechanisms called **Concurrency Control Schemes.**

When multiple transactions are executed simultaneously, it is highly important to control the concurrency of transactions. We have concurrency control protocols to ensure atomicity, isolation, and serializability of concurrent transactions.

Concurrency control protocols can be broadly divided into three categories –**Lock based protocols, time stamp based protocols and validation based protocols**

**Topic: Lock-Based Protocols:**

* A lock is a mechanism to control concurrent access to a data item, that is while one transaction is accessing a data item , no other transaction can modify that data item
* Data items can be locked in two modes :

1. ***shared (S) mode*.** It is also known as a Read-only lock. In a shared lock, the data item can only read by the transaction.S-lock is requested using lock-S instruction.
2. ***exclusive (X) mode*** In this the data item can be both reads as well as written by the transaction**.** X-lock is requested using lock-X instruction

**Lock-compatibility matrix**



* Lock requests are made to the concurrency-control manager by the programmer. Transaction can proceed only after request is granted.
* A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
* Any number of transactions can hold shared locks on an item,But if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
* If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.
* A locking protocol is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

**Example of a transaction performing locking:**

***T1*: lock-S*(A)*;**

**read*(A)*;**

**unlock*(A)*;**

**lock-S*(B)*;**

**read*(B)*;**

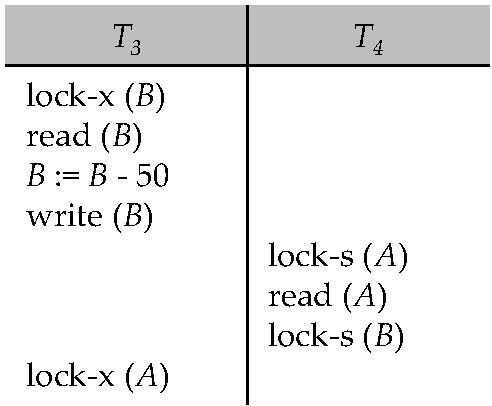
**unlock*(B)*;**

**display*(A+B)***

* Locking as above is not sufficient to guarantee serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.

**Pitfalls of lock-based protocols:**

Consider the below partial schedule



* In above neither *T3* nor *T4* can make progress — executing lock-S*(B)* causes *T4* to wait for *T3* to release its lock on *B*, while executing lock-X*(A)* causes *T3* to wait for *T4* to release its lock on *A*.
* Such a situation is called **Deadlock,** to handle a deadlock one of T3 or T4 must be rolled back and its lock released.
* **Starvation** is also possible if concurrency control manager is badly designed. For example-a transaction may be waiting for an lock-X on an data item,while a sequence of other transactions request and are granted an lock-S on the same data item. The same transaction is rolled back due to deadlock, this situation is called starvation.

**The Two-Phase Locking Protocol:**

This protocol ensures conflict-serializable schedules and requires each transaction issue lock and unlock requests in two phases:

Phase 1: Growing Phase

* + Transaction may obtain locks
  + Transaction may not release locks

Phase 2: Shrinking Phase

* + Transaction may release locks
  + Transaction may not obtain locks
* Initially, a transaction is in the growing phase that is transactions acquires locks as needed , then it enters the shrinking phase, and it can issue no more lock requests.
* The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e., the point where a transaction acquired its final lock).
* There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.
* However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:
* Given a transaction *T*i that does not follow two-phase locking, we can find a transaction *Tj* that uses two-phase locking, and a schedule for *Ti* and *Tj* that is not conflict serializable.

**Lock Conversions**

* A refinement of basic two-phase locking protocol ,in which lock conversions are allowed.
* Upgrade:
  + Converting from shared to exclusive modes
  + This takes place only in growing phase
* Downgrade
  + Converting from exclusive to shared mode
  + This takes place only in shrinking phase

Example:

T1 T2

1 Lock-s(A)

2 Lock-s(A)

3 Lock-X(B)

4 Unlock-s(A)

5 Lock-x(C)

6 Unlock-X(B)

7 Unlock-s(A)

8 Unlock-x(C)

For Transaction T1: Growing phase: from step 1-3,Shrinking phase: from step 4-6,Lock point is at step 3

For Transaction T2: Growing phase: from step 2-5, Shrinking phase: from step 7-8,Lock point is at step 5

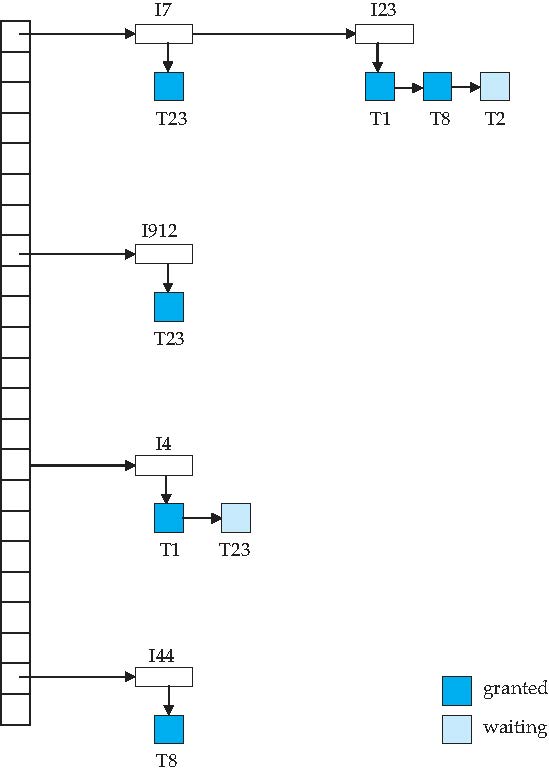
**Strict Two-phase locking (Strict-2PL)**

* The first phase of Strict-2PL is similar to 2PL. In the first phase, after acquiring all the locks, the transaction continues to execute normally.
* The only difference between 2PL and strict 2PL is that Strict-2PL does not release a lock after using it.
* Strict-2PL waits until the whole transaction to commit, and then it releases all the locks at a time.
* Strict-2PL protocol does not have shrinking phase of lock release.
* Rigorous two-phase locking is even stricter. Here, *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

**Implementation of Locking:**

* A lock manager can be implemented as a separate process to which transactions send lock and unlock requests
* The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
* The requesting transaction waits until its request is answered
* The lock manager maintains a data-structure called a lock table to record granted locks and pending requests
* The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

**Lock Table:**



Dark blue rectangles indicate granted locks; light blue indicate waiting requests

* Lock table also records the type of lock granted or requested.
* New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks.
* Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted.
* If transaction aborts, all waiting or granted requests of the transaction are deleted.
* lock manager may keep a list of locks held by each transaction, to implement this efficiently.

**Topic: Timestamp-Based Protocols**

* Each transaction is issued a timestamp when it enters the system. If an old transaction *Ti* has time-stamp TS(*Ti*), a new transaction *Tj* is assigned time-stamp TS(*Tj*) such that TS(*Ti*) <TS(*Tj*).
* The protocol manages concurrent execution such that the time-stamps determine the serializability order.
* In order to assure such behavior, the protocol maintains for each data *Q* two timestamp values:
  + W-timestamp(*Q*) is the largest time-stamp of any transaction that executed write(*Q*) successfully.
  + R-timestamp(*Q*) is the largest time-stamp of any transaction that executed read(*Q*) successfully.
* The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order.

--Suppose a transaction Ti issues a read(*Q*)

* 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.
* If TS(*Ti*) ≥W-timestamp(*Q*), then the read operation is executed, and R-timestamp(*Q*) is set to max(R-timestamp(*Q*), TS(*Ti*)).

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

* If TS(*Ti*) < R-timestamp(*Q*), then the value of *Q* that *Ti* is producing was needed previously, and the system assumed that value would never be produced.

Hence, the write operation is rejected, and *Ti* is rolled back.

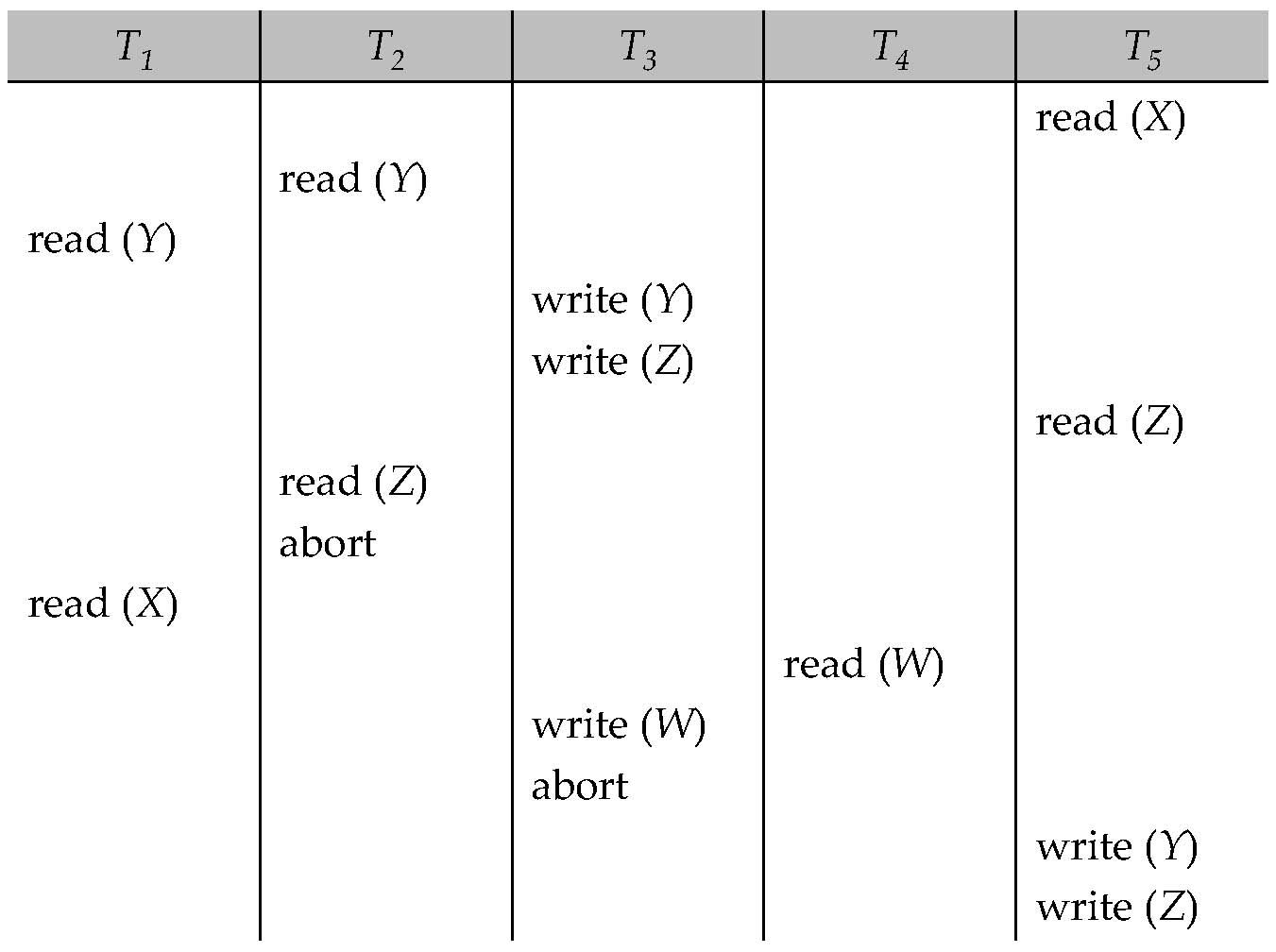
* If TS(*Ti*) < W-timestamp(*Q*), then *Ti* is attempting to write an obsolete value of *Q*.

Hence, this write operation is rejected, and *Ti* is rolled back.

* Otherwise, the write operation is executed, and W-timestamp(*Q*) is set to TS(*Ti*).

Example Use of the Protocol:

* A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

****

**Correctness of Timestamp-Ordering Protocol**

* The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:

Transaction with larger TS

Transaction with smaller TS

Thus, there will be no cycles in the precedence graph

* Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
* But the schedule may not be cascade-free, and may not even be recoverable.

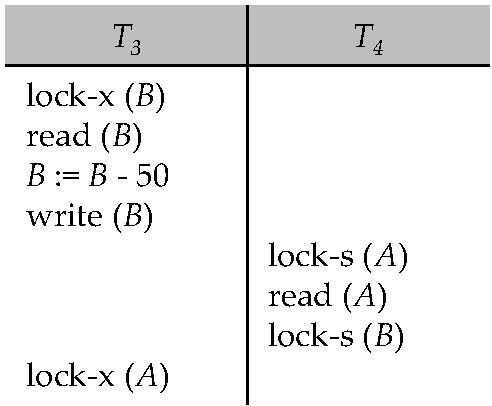
Recoverability and Cascade Freedom:

* Problem with timestamp-ordering protocol:
  + Suppose *Ti* aborts, but *Tj* has read a data item written by *Ti*
  + Then *Tj*must abort; if *Tj*had been allowed to commit earlier, the schedule is not recoverable.
  + Further, any transaction that has read a data item written by *Tj* must abort
  + This can lead to cascading rollback --- that is, a chain of rollbacks
* Solution 1:
  + A transaction is structured such that its writes are all performed at the end of its processing
  + All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written
  + A transaction that aborts is restarted with a new timestamp
* Solution 2: Limited form of locking: wait for data to be committed before reading it
* Solution 3: Use commit dependencies to ensure recoverability

**Topic: Deadlocks:**

A deadlock is a condition where two or more transactions are waiting indefinitely for one another to give up locks. Deadlock is said to be one of the most feared complications in DBMS as no task ever gets finished and is in waiting state forever.

Consider the partial schedule



* Neither *T3* nor *T4* can make progress — executing lock-S*(B)* causes *T4* to wait for *T3* to release its lock on *B*, while executing lock-X*(A)* causes *T3* to wait for *T4* to release its lock on *A*.
* Such a situation is called a deadlock.
  + To handle a deadlock one of *T3* or *T4* must be rolled back and its locks released.

**Topic: Deadlock Handling:**

* System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.

There are two methods to handle deadlock

1.Deadlock Prevention

2.Deadlock Detection and Recovery

* *Deadlock prevention* protocols ensure that the system will *never* enter into a deadlock state
* *Deadlock Detection and Recovery* protocols allows the system to enter into a deadlock state and then try to recover it

**Deadlock Prevention:**

*Deadlock prevention* protocols ensure that the system will *never* enter into a deadlock state.

Some prevention strategies :

1.Require that each transaction locks all its data items before it begins execution(predeclaration).

Disadvantages:

a. It is often hard to predict, what data items need to be locked before the transaction

b. data item utilization may be very low

2.Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order.

* Instead of partial put the total ordering with two phase locking.

More Deadlock Prevention Strategies:

Following schemes use transaction timestamps for deadlock prevention alone.

1.wait-die scheme — non-preemptive

* + older transaction may wait for younger one to release data item. (older means smaller timestamp) Younger transactions never wait for older ones; they are rolled back instead.
  + a transaction may die several times before acquiring needed data item

2.wound-wait scheme — preemptive

* + older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
  + may be fewer rollbacks than *wait-die* scheme.
  + Both in *wait-die* and in *wound-wait* schemes, a rolled back transactions is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.

3.Timeout-Based Schemes:

* + a transaction waits for a lock only for a specified amount of time. If the lock has not been granted within that time, the transaction is rolled back and restarted, thus, deadlocks are not possible.
  + simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

**Deadlock Detection:**

When a transaction waits indefinately to obtain a lock, The database managememt system should detect whether the transaction is involved in a deadlock or not.

Deadlocks can be described precisely in terms of a directed graph called wait-for graph.

**Wait-for-graph** is one of the methods for detecting the deadlock situation. This method is suitable for smaller database. In this method a graph is drawn based on the transaction and their lock on the resource. If the graph created has a closed loop or a cycle, then there is a deadlock.

The wait for the graph is maintained by the system for every transaction which is waiting for some data held by the others. The system keeps checking the graph if there is any cycle in the graph.

**Wait-for graph without a cycle**

**Wait-for graph with a cycle**

**Deadlock Recovery:**

When deadlock is detected, the system must recover from the deadlock.

The most common solution is to rollback the transaction to break the deadlock.The following three actions should be taken:

* 1. Selection of victim: Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.

The points to remember for determining the rollback of transaction are—how many data items the transaction has used?,how many more data items the transaction needs for it to complete?

* 1. Rollback –Once we have decided that a particular transaction must be rolled back, we must determine how far to roll back transaction
     + Total rollback: Abort the transaction and then restart it.However it is efficient to roll back the transaction to break deadlock.
     + The deadlock detection mechanism should decide which locks the selected transaction needsto release in order to break the deadlock
  2. Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation