# Concurrency Control Techniques

# How to solve Interference Between Concurrent Transactions

- **Recall:**
- The objective of concurrency control protocol is to schedule transactions in such a way as to avoid any interference between them.
- When there are multiple transactions that are running in a concurrent manner and the order of operation is needed to be set so that the operations do not overlap each other, Scheduling is brought into play and the transactions are timed accordingly.

# How to solve Interference Between Concurrent Transactions

- ***A quick solution*** is to allow only one transaction to be executed and committed before another transaction begins to execute.
- However, the aim of multi-user DBMS is to maximize the degree of concurrency or parallelism in the system without interference among the transactions.

# Preventing Anomalies Through Concurrency Control Techniques

- Serializability can be achieved in several ways.
- There are two main concurrency control techniques:
  - Locking Method and
  - Timestamp Method
- The two methods are essentially conservative (or pessimist) approaches in the sense that they cause transactions to be delayed incase they conflict with other transactions at some time in the future.

# Lock-Based Protocols

- 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.

- **Locking** is a procedure used to control concurrent access to data. When one transaction is accessing the database, a lock may deny access to other transactions to prevent incorrect/inconsistent results.

- Transactions request for locks from the scheduler (called lock manager)

# Lock-Based Protocols

- Data items can be locked in two modes :
  1. **Shared lock** - If a transaction wants to read an object, it must first request a **shared lock** (S mode) on that object.

  2. **Exclusive lock** - If a transaction wants to modify an object, it must first request an **exclusive lock** (X mode) on that object

- Since read operations can not conflict, it is permissible for more than one transaction to hold shared locks simultaneously on the same item

- Exclusive lock prevents another transaction from modifying the item or even reading it.

# Lock-Based Protocols

## Lock Manager

- A Lock manager can be implemented as a separate process to which transactions send lock and unlock requests

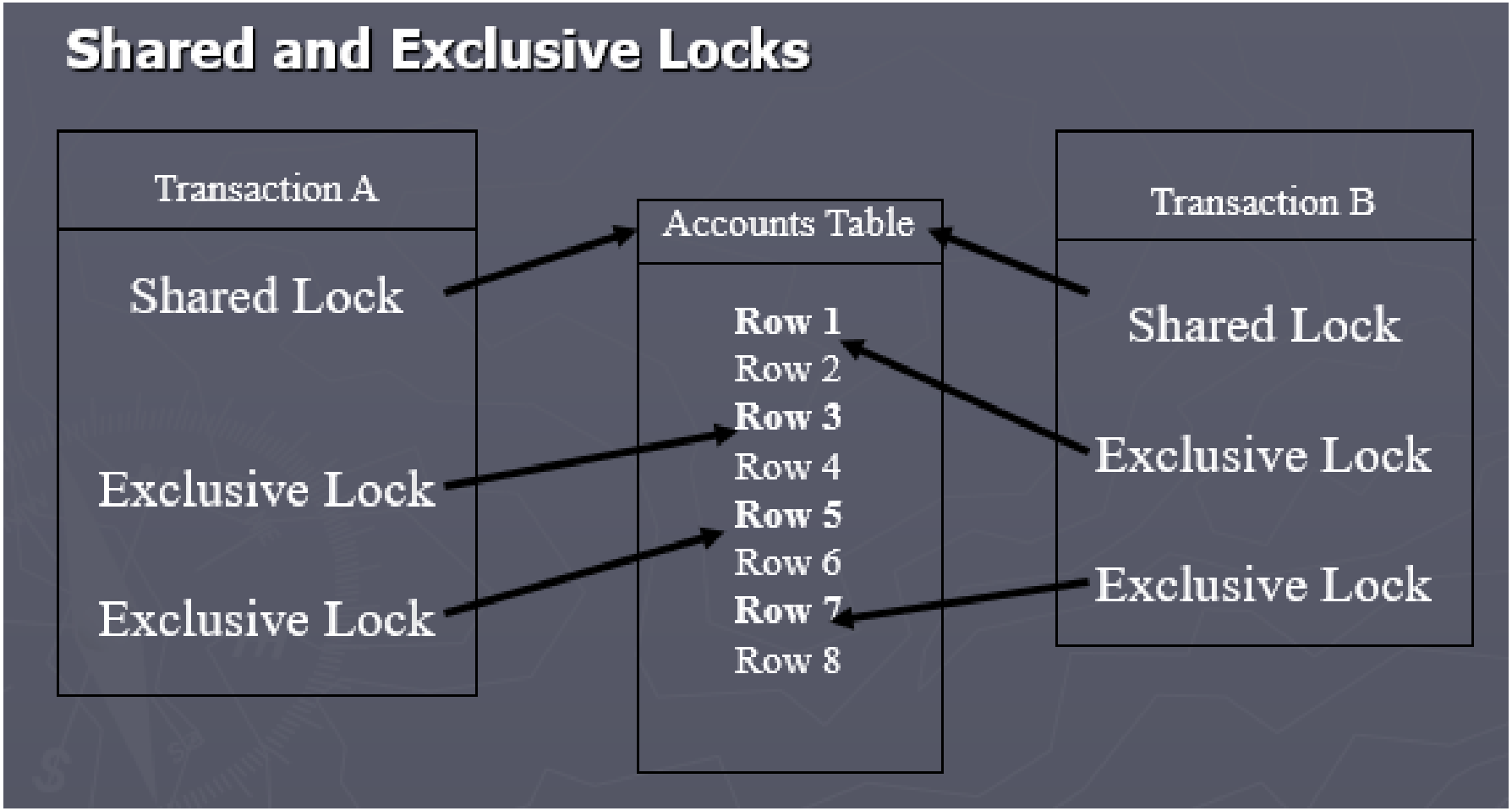
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)

- 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

# Lock-Based Protocols

## Lock Manager



# Lock-Based Protocols

- **Lock-compatibility matrix**

| | S | X |
|---|---|---|
| S | true | false |
| X | false | false |

- 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 lock 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.

# Lock-Based Protocols

## Locking Procedure

1. Any transaction that needs to access a data item must first lock the item by requesting a shared lock for read only access or an exclusive lock for both read and write access.
2. If an item is not already locked by another transaction, the lock will be granted.
3. 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 has already a shared lock on it, the request will be granted, otherwise the transaction must wait until the existing lock is released.
4. 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.

# Lock-Based Protocols

## Example

Example of a transaction performing locking:

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

***Note:*** 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.

# Lock-Based Protocols

## Two Phase (2PL) Locking Protocol

- A transaction follows the two-phase locking protocol if all locking operations precede the first unlock operation in the transaction.
- Every transaction can be divided into two phases:
  1. **Growing phase** in which it acquires all the locks needed but cannot release any locks.
  2. **Shrinking phase** in which it releases its locks but cannot acquire any new locks.

# Lock-Based Protocols

## Two Phase (2PL) Locking Protocol

The rules are:

1. 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.
2. Once the transaction releases a lock, it can never acquire any new locks.
3. Upgrading of locks can only take place in the growing phase.
4. Downgrading can only take place during the shrinking phase.

**NOTE:** Lock point is the point where a transaction acquired its final lock.

# Lock-Based Protocols

## Two Phase (2PL) Locking Protocol

- Two-phase locking with lock conversions:

  **First Phase:**
  - can acquire a lock-S on item
  - can acquire a lock-X on item
  - can convert a lock-S to a lock-X (**upgrade**)

  **Second Phase:**
  - can release a lock-S
  - can release a lock-X
  - can convert a lock-X to a lock-S (**downgrade**)

- This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.

# Lock-Based Protocols

## Two Phase (2PL) Locking Protocol

**Example :**

| $T_5$ | $T_6$ | $T_7$ |
|---|---|---|
| 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($A$) |

# Lock-Based Protocols

## Types of two Phase (2PL) Locking Protocol

- There are many protocols derived from 2PL :
  - **Strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits.
  - **Rigorous two-phase locking** is even stricter: here all locks (shared and exclusive) are held till commit. In this protocol transactions can be serialized in the order in which they commit.
  - **Graph-based protocol** : we fix an order of accessing data. If a transaction has to update Row2 and read Row1, it has to access these data in a predefined order.

# Preventing The Lost Update Problem using 2PL

***Example II:*** Transaction $T_1$ is executed concurrently with transaction $T_2$.

| Time | $T_1$ | $T_2$ | $Bal_x$ |
|---|---|---|---|
| $t_1$ | - | begin_transaction | 100 |
| $t_2$ | begin_transaction | read($bal_x$) | 100 |
| $t_3$ | read($bal_x$) | $bal_x = bal_x + 100$ | 100 |
| $t_4$ | $bal_x = bal_x - 10$ | write($bal_x$) | 200 |
| $t_5$ | write($bal_x$) | Commmit | 90 |
| $t_6$ | Commmit | - | 90 |
| | | | |

- Transaction $T_1$ updates the same record updated earlier by $T_2$ at time $t_5$ on the basis of values read at $t_3$ .
- **Question: How do we solve the lost update problem?**

# 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 | Commmit/unlock($bal_x$) | 200 |
| $t_7$ | read($bal_x$) | | 200 |
| $t_8$ | $bal_x = bal_x - 10$ | | 200 |
| $t_9$ | write($bal_x$) | | 190 |
| $t_{10}$ | Commmit/unlock($bal_x$) | | 190 |

# Preventing the Uncommitted Dependency Problem using 2PL

**Example I:** Consider this situation:

| Time | $T_3$ | $T_4$ | $Bal_x$ |
|---|---|---|---|
| $t_1$ | - | begin_transaction | 100 |
| $t_2$ | - | read($bal_x$) | 100 |
| $t_3$ | - | $bal_x = bal_x + 100$ | 100 |
| $t_4$ | begin_transaction | write($bal_x$) | 200 |
| $t_5$ | read($bal_x$) | : | 200 |
| $t_6$ | $bal_x = bal_x - 10$ | rollback | 100 |
| $t_7$ | write($bal_x$) | - | 190 |
| $t_8$ | commmit | - | 190 |

**Question: How do we solve the uncommitted dependency problem?**

# 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 | $bal_x = bal_x + 100$ | 200 |
| $t_5$ | 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 |
| $t_{10}$ | commmit/unlock($bal_x$) | | 90 |

# Preventing Inconsistent Analysis Problem using 2PL

| Time | Transaction A | Transaction B | Acc1 | Acc2 | Acc3 | Sum |
|---|---|---|---|---|---|---|
| $t_1$ | Read Acc1 | Read Acc1 | 100 | 50 | 25 | 0 |
| $t_2$ | Acc1 - 10 | Sum + Acc1 | 100 | 50 | 25 | 100 |
| $t_3$ | Write Acc1 | Read Acc2 | 90 | 50 | 25 | 100 |
| $t_4$ | Read Acc3 | Sum + Acc2 | 90 | 50 | 25 | 150 |
| $t_5$ | Acc3 + 10 | - | 90 | 50 | 25 | 150 |
| $t_6$ | Write Acc3 | - | 90 | 50 | 35 | 150 |
| $t_6$ | Commit | Read Acc3 | 90 | 50 | 35 | 150 |
| $t_7$ | - | Sum + Acc3 | 90 | 50 | 35 | 185 |
| $t_8$ | - | Commit | 90 | 50 | 35 | 185 |

- Sum = 185 not 175!!

**Question: How do we solve the inconsistent analysis problem?**

# Assignment I

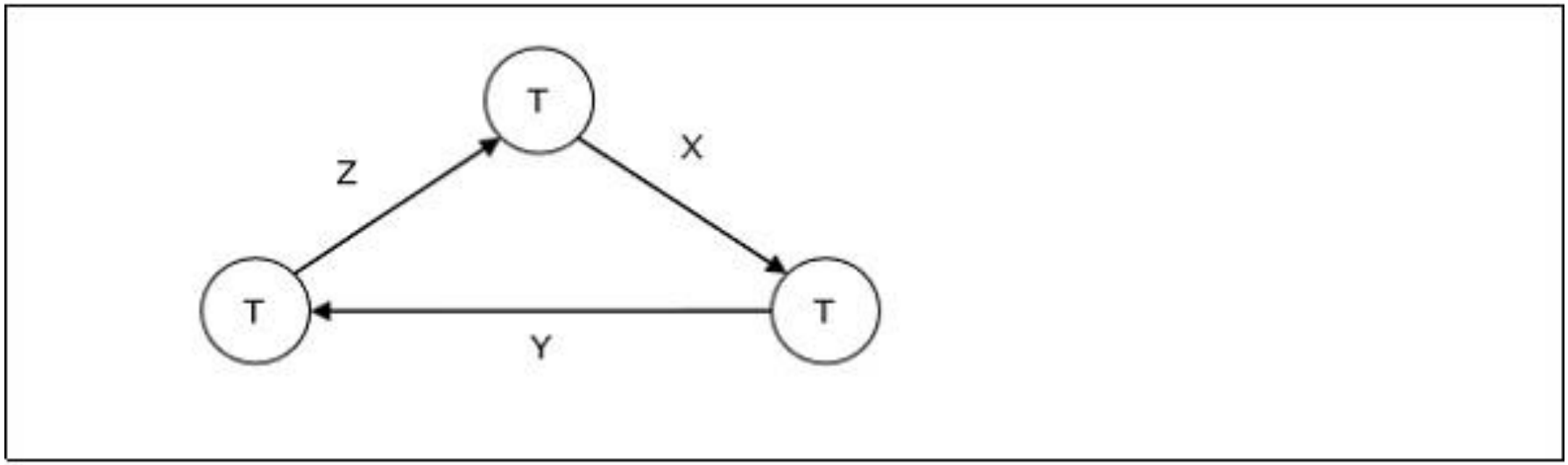
Using 2PL, provide a solution to the inconsistent analysis problem discussed earlier.

# Deadlock

- A deadlock is an impasse that may result when two (or more) transactions are each waiting for locks to be released that are held by the other.
- Neither transactions in a deadlocked system can continue because each is waiting for a lock it cannot obtain until the other completes.
- Once a deadlock occurs, the applications involved cannot resolve the problem, instead the DBMS has to recognise that deadlock exists and break the deadlock in some way.
- The only way to break a deadlock is to abort one or more transactions.

# Deadlock

- A deadlock can be indicated by a cycle in the wait-for-graph.
  - This is a directed graph in which the vertices denote transactions and the edges denote waits for data items.
- **Example1:** In the wait-for-graph below, transaction T1 is waiting for data item X which is locked by T3. T3 is waiting for Y which is locked by T2 and T2 is waiting for Z which is locked by T1. Hence, a waiting cycle is formed, and none of the transactions can proceed executing.



# Deadlocks

## Example 1

- In this partial schedule, neither $T_3$ nor $T_4$ can make progress — executing **lock-S**(*B*) causes $T_4$ to wait for $T_3$ to release its lock on *B*, while executing **lock-X**(*A*) causes $T_3$ to wait for $T_4$ to release its lock on *A*.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of $T_3$ or $T_4$ must be rolled back and its locks released.

| $T_3$ | $T_4$ |
|---|---|
| lock-x (*B*) | |
| read (*B*) | |
| *B* := *B* - 50 | |
| write (*B*) | |
| | lock-s (*A*) |
| | read (*A*) |
| | lock-s (*B*) |
| lock-x (*A*) | |

# Deadlock

## Example 3

| Time | $T_3$ | $T_4$ |
|---|---|---|
| $t_1$ | begin_transaction | |
| $t_2$ | Write_lock($bal_x$) | begin_transaction |
| $t_3$ | read($bal_x$) | Write_lock($bal_y$) |
| $t_4$ | $bal_x = bal_x - 10$ | read($bal_y$) |
| $t_5$ | write($bal_x$) | $bal_y = bal_y + 100$ |
| $t_6$ | Write_lock($bal_y$) | write($bal_y$) |
| $t_7$ | wait | Write_lock($bal_x$) |
| $t_8$ | wait | wait |
| $t_9$ | wait | wait |
| $t_{10}$ | wait | wait |
| $t_{11}$ | : | : |

# Deadlocks

- Two-phase locking *does not* ensure freedom from deadlocks.
- In addition to deadlocks, there is a possibility of **starvation.**
- **Starvation** occurs if the concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

# Deadlocks

- The potential for deadlock exists in most locking protocols.
- When a deadlock occurs there is a possibility of cascading roll-backs.
- Cascading roll-back is possible under two-phase locking.
- To avoid this, we use:

1. **Strict two-phase locking** which is a modified protocol in which a transaction must hold all its exclusive locks till it commits/aborts.
2. **Rigorous two-phase locking** which is even stricter.
   - all locks are held till commit/abort.
   - In this protocol transactions can be serialized in the order in which they commit.

# Deadlock Handling in Centralized Systems

- There are three classical approaches for deadlock handling, namely –
  - Timeouts
  - Deadlock prevention.
  - Deadlock detection and removal

# Dealing with Deadlocks
## Timeouts

- A transaction that requests for a lock will wait for only a system-defined period of time. If the lock has not been granted within this period, the lock requests times out and the transaction is rolled back and restarted,

- DBMS assumes the transaction may be deadlocked, even if it may not be, and it aborts and automatically restarts the transaction.

- simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

# Dealing with Deadlocks
## Deadlock Prevention

- The deadlock prevention approach does not allow any transaction to acquire locks that will lead to deadlocks.
  - The DBMS looks ahead to determine if a transaction would cause deadlock, and never allows deadlock to occur.
  - The convention is that when more than one transactions request for locking the same data item, only one of them is granted the lock.

# Dealing with Deadlocks

## Deadlock Prevention

- Some prevention strategies :
  - One of the most popular deadlock prevention methods is pre-acquisition of all the locks.
    - In this method, a transaction acquires all the locks before starting to execute and retains the locks for the entire duration of transaction.
    - If another transaction needs any of the already acquired locks, it has to wait until all the locks it needs are available.
  - 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.

# Dealing with Deadlocks
## Deadlock Prevention

- The deadlock prevention/avoidance approach handles deadlocks before they occur.
  - It analyzes the transactions and the locks to determine whether or not waiting leads to a deadlock.
- Transactions start executing and request data items that they need to lock.
- The lock manager checks whether the lock is available.
- If it is available, the lock manager allocates the data item and the transaction acquires the lock.

(Continued…)

# Dealing with Deadlocks
## Deadlock Prevention

- However, if the item is locked by some other transaction in incompatible mode, the lock manager runs an algorithm to test whether keeping the transaction in waiting state will cause a deadlock or not.

- Accordingly, the algorithm decides whether the transaction can wait or one of the transactions should be aborted.

# Dealing with Deadlocks
## Deadlock Prevention

- There are two algorithms for this purpose that use transaction timestamps for the sake of deadlock avoidance/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.

# Dealing with Deadlocks
## Deadlock Prevention

- **Example:** Let us assume that there are two transactions, T1 and T2, where T1 tries to lock a data item which is already locked by T2. The algorithms are as follows –
  - **Wait-Die** – If T1 is older than T2, T1 is allowed to wait. Otherwise, if T1 is younger than T2, T1 is aborted and later restarted.
  - **Wound-Wait** – If T1 is older than T2, T2 is aborted and later restarted. Otherwise, if T1 is younger than T2, T1 is allowed to wait.

# Dealing with Deadlocks
## Deadlock Detection and Removal

- The deadlock detection and removal approach runs a deadlock detection algorithm periodically and removes deadlock in case there is one.
- This approach does not check for deadlock when a transaction places a request for a lock.
  - When a transaction requests a lock, the lock manager checks whether it is available.
  - If it is available, the transaction is allowed to lock the data item; otherwise the transaction is allowed to wait.
- Since there are no precautions while granting lock requests, some of the transactions may be deadlocked.

# Dealing with Deadlocks

## Deadlock Detection and Removal

- To detect deadlocks, the lock manager periodically checks if the wait-for graph has cycles.
- If the system is deadlocked, the lock manager chooses a victim transaction from each cycle.
- The victim is aborted and rolled back; and then restarted later.

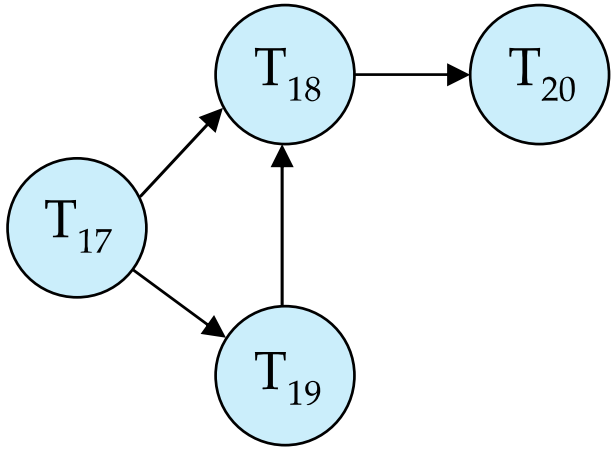
# Dealing with Deadlocks
## Deadlock Detection and Removal

- Some of the methods used for victim selection are –
  - Choose the youngest transaction.
  - Choose the transaction with fewest data items.
  - Choose the transaction that has performed least number of updates.
  - Choose the transaction having least restart overhead.
  - Choose the transaction which is common to two or more cycles.

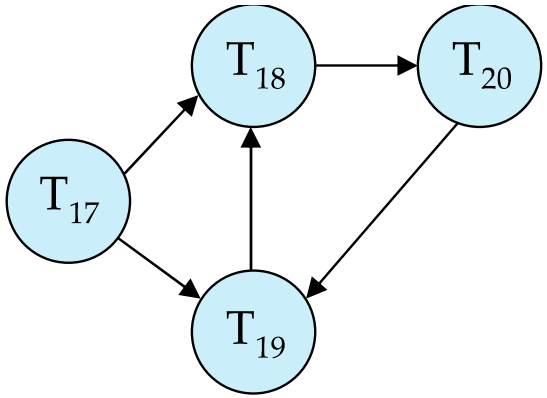
- This approach is primarily suited for systems having transactions low and where fast response to lock requests is needed.

# Dealing with Deadlocks

## Deadlock Detection




Wait-for graph without a cycle




Wait-for graph with a cycle

# Revision Questions

1. Discuss the differences between Concurrency Control with Locking and Concurrency Control without Locking.

2. Discuss the differences between Pessimistic concurrency control and Optimistic concurrency control.

3. Describe the basic timestamp ordering protocol for concurrency control, and how does it differ from locking based protocols.

# Optimistic and Pessimistic Concurrency Control

- Transactional isolation is usually implemented by locking whatever is accessed in a transaction.
- There are two different approaches to transactional locking:
  - Pessimistic locking
  - Optimistic locking

# Pessimistic Locking

- Pessimistic locking is called "pessimistic" because the system assumes the worst:
  - It assumes that two or more users will want to update the same record at the same time, and then prevents that possibility by locking the record, no matter how unlikely conflicts actually are.
  - Pessimistic locking assumes that data will be changed by another transaction and so locks it.
  - Pessimistic locking locks the records as soon as it selects rows to update.
  - The pessimistic locking strategy guarantees the changes are made safely and consistently.

# Pessimistic Locking

- The disadvantage of pessimistic locking is that a resource is locked from the time it is first accessed in a transaction until the transaction is finished, making it inaccessible to other transactions during that time.
- If most transactions simply look at the resource and never change it, an exclusive lock may be overkill as it may cause lock contention, and optimistic locking may be a better approach.

# Optimistic Locking

- Optimistic locking assumes that although conflicts are possible, they will be very rare.
  - Instead of locking every record every time that it is used, the system merely looks for indications that two users actually did try to update the same record at the same time.
  - Optimistic locking locks the record only when updating takes place.
  - It ensures that the locks are held between selecting, updating, or deleting rows.

# Optimistic Locking

- Optimistic process needs a way to ensure that the changes to data are not performed between the time of being read and being altered.
- This is achieved using **version numbers**, **timestamps**, **hashing**, etc.
- The primary advantage of optimistic locking is , it minimizes the time for which a given resource is unavailable which is used by another transaction and in this way it more scalable locking alternative

# Pessimistic VS Optimistic Approaches

- Locking is pessimistic
  - Use blocking to avoid conflicts
  - Overhead of locking even if contention is low
- Optimistic concurrency control
  - Assume that most transactions do not conflict
  - Let them execute as much as possible
  - If it turns out that they conflict, abort and restart
  - Timestamp-based

# Pessimistic vs Optimistic

- Overhead of locking vs overhead of validation and copying private space
- Blocking versus Aborts and Restarts
- Note
  - Locking has better throughput for environments with medium-to-high contention
  - Optimistic concurrency control is better when resource utilization is low enough

# Optimistic: Sketch of Protocol

- Read phase: transaction executes, reads from the database, and writes to a private space
- Validate phase: DBMS checks for conflicts with other transactions; if conflict is possible, abort and restart
  - Requires maintaining a list of objects read and written by each transaction
- Write phase: copy changes in the private space to the database

# Timestamp-based

- Associate each database object with a read timestamp and a write timestamp
- Assign a timestamp to each transaction (Timestamp order is commit order)
- When transaction reads/writes an object, check the object's timestamp for conflict with a younger transaction; if so, abort and restart

# References


- https://www.geeksforgeeks.org/types-of-schedules-in-dbms/
- http://www.ccs.neu.edu/home/kathleen/classes/cs3200/9-Transactions.pdf