# Database Management Systems

## UNIT IV



## Learning Objectives

**Transaction Processing Concept:**

1. Transaction System
2. Testing of Serializability
3. Serializability of Schedules
4. Conflict & View Serializable Schedule
5. Recoverability
6. Recovery from Transaction Failures
7. Log-based Recovery
8. Checkpoints
9. Deadlock Handling

**Concurrency Control Technique:**

1. Concurrency Control
2. Locking Techniques
3. Timestamp Ordering
4. Granularity of data items
5. Recovery from catastrophic failures
6. Concepts of OODBMS
7. Overview of Database Security Concepts



## Transaction System

- The operations performed in a transaction include **one or more of database operations** like insert, delete, update or retrieve data.
- It is an atomic process that is either performed into completion **entirely or is not performed at all.**
- A transaction involving **only data retrieval** without any data update is called **read-only transaction**.

**ACID** Properties

## Transaction System

- Examples of these systems include airline reservations, banking, credit card processing etc.
- These system requires high availability and fast response time for hundreds of concurrent users.
- **Single User and Multiple Users.**
- Concurrently access the database.
- Multiprogramming



## Transaction System

- A transaction is a *unit* of program execution that accesses and possibly updates various data items.
- It includes insertion, deletion, modification and retrieval operations.
- The boundary of transactions can be defined with the help of begin transaction and end transaction.

- E.g. transaction to transfer ₹50 from account A to account B:
  1. read(*A*)
  2. *A* := *A* – 50
  3. write(*A*)
  4. read(*B*)
  5. *B* := *B* + 50
  6. write(*B*)



## Transaction System (Lost Update)

| Schedule 1 | Transaction $T_3$ | Transaction $T_4$ | Value of $A$ |
|---|---|---|---|
| | | Read($A$) | 200 |
| | | $A := A * 1.1$ | |
| Read($A$) | Read($A$) | | |
| $A := A + 10$ | $A := A + 10$ | | |
| Write($A$) | Write($A$) | | 210 |
| Write($A$) | | Write($A$) | 220 |

(a)

| Schedule 2 | Transaction $T_3$ | Transaction $T_4$ | Value of $A$ |
|---|---|---|---|
| | | | 200 |
| Read($A$) | Read($A$) | | |
| $A := A + 10$ | $A := A + 10$ | | |
| Read($A$) | | Read($A$) | |
| $A := A * 1.1$ | | $A := A * 1.1$ | |
| Write($A$) | | Write($A$) | 220 |
| Write($A$) | Write($A$) | | 210 |

(b)

## Transaction System (Inconsistent Reads)

| | Schedule | Transaction $T_5$ | Transaction $T_6$ | Value of Database items A | B | Sum |
|---|---|---|---|---|---|---|
| | **Read**(*A*) | **Read**(*A*) | | 500 | 1100 | — |
| | *Sum* := 0 | | Sum := 0 | | | 0 |
| T | **Read** (*A*) | | **Read** (*A*) | | | |
| i | *A* := *A* − 100 | *A* := *A* − 100 | | | | |
| m | **Write**(*A*) | **Write**(*A*) | | 400 | | |
| e | *Sum* := *Sum* + *A* | | *Sum* := *Sum* + *A* | | | 500 |
| | **Read**(*B*) | **Read**(*B*) | | | | |
| | *B* := B + 100 | *B* := *B* + 100 | | | | |
| | **Write**(*B*) | **Write**(*B*) | | | 1100 | |
| | **Read**(*B*) | | **Read**(*B*) | | | |
| | *Sum* := *Sum* + *B* | | *Sum* := *Sum* + *B* | | | |
| | **Write**(*Sum*) | | **Write**(*Sum*) | | | 1600 |



## Serializability of Schedules

- In a Serial Execution, each transaction runs to completion before any statements from any other transaction are executed.

| | Schedule A | Transaction $T_3$ | Transaction $T_4$ |
|---|---|---|---|
| | **Read**(*A*) | **Read**(*A*) | |
| T | *A* := *A* + 10 | *A* := *A* + 10 | |
| i | **Write**(*A*) | **Write**(*A*) | |
| m | **Read**(*A*) | | **Read**(*A*) |
| e | *A* := *A* * 1.1 | | *A* := *A* * 1.1 |
| | **Write**(*A*) | | **Write**(*A*) |

(a)

| | Schedule B | Transaction $T_3$ | Transaction $T_4$ |
|---|---|---|---|
| | **Read**(*A*) | | **Read**(*A*) |
| T | *A* := *A* * 1.1 | | *A* := *A* * 1.1 |
| i | **Write**(*A*) | | **Write**(*A*) |
| m | **Read**(*A*) | **Read**(*A*) | |
| e | *A* := *A* + 10 | *A* := *A* + 10 | |
| | **Write**(*A*) | **Write**(*A*) | |

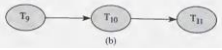
(b)



## Precedence Graph

| | Schedule | Transaction $T_9$ | Transaction $T_{10}$ | Transaction $T_{11}$ |
|---|---|---|---|---|
| | **Read**(*A*) | **Read**(*A*) | | |
| | *A* := $f_1$(*A*) | *A* := $f_1$(*A*) | | |
| T | **Write**(*A*) | **Write**(*A*) | | |
| i | **Read**(*A*) | | **Read**(*A*) | |
| m | *A* := $f_2$(*A*) | | *A* := $f_2$(*A*) | |
| e | **Write**(*A*) | | **Write**(*A*) | |
| | **Read**(*B*) | | **Read**(*B*) | |
| | *A* := $f_3$(*B*) | | *B* := $f_3$(*B*) | |
| | **Write**(*B*) | | **Write**(*B*) | |
| | **Read**(*B*) | | | **Read**(*B*) |
| | *A* := $f_4$(*B*) | | | *B* := $f_4$(*B*) |
| | **Write**(*B*) | | | **Write**(*B*) |

(a)



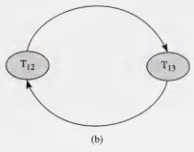

(b)

# Precedence Graph

| Schedule | Transaction $T_{12}$ | Transaction $T_{13}$ |
|---|---|---|
| **Read**$(A)$ | **Read**$(A)$ | |
| $A := f_1(A)$ | $A := f_1(A)$ | |
| **Read**$(A)$ | | **Read**$(A)$ |
| $A := f_2(A)$ | | $A := f_2(A)$ |
| **Write**$(A)$ | | **Write**$(A)$ |
| **Write**$(A)$ | **Write**$(A)$ | |

Time ↓

(a)




(b)



# Testing of Serializability

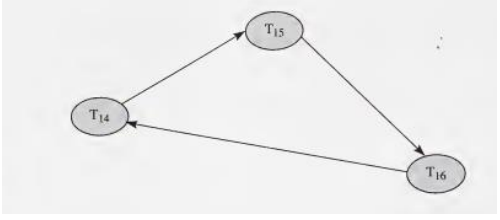
- The precedence graph for serializable schedule S must be acyclic, hence it can be converted to a serial schedule

| Schedule | Transaction $T_{14}$ | Transaction $T_{15}$ | Transaction $T_{16}$ |
|---|---|---|---|
| **Read**$(A)$ | **Read**$(A)$ | | |
| **Read**$(B)$ | | **Read**$(B)$ | |
| $A := f_1(A)$ | $A := f_1(A)$ | | |
| **Read**$(C)$ | | | **Read**$(C)$ |
| $B := f_2(B)$ | | $B := f_2(B)$ | |
| **Write**$(B)$ | | **Write**$(B)$ | |
| $C := f_3(C)$ | | | $C := f_3(C)$ |
| **Write**$(C)$ | | | **Write**(C) |
| **Write**$(A)$ | **Write**$(A)$ | | |
| **Read**$(B)$ | | | **Read**$(B)$ |
| **Read**$(A)$ | | **Read**$(A)$ | |
| $A := f_4(A)$ | | $A := f_4(A)$ | |
| **Read**$(C)$ | **Read**$(C)$ | | |
| **Write**$(A)$ | | **Write**$(A)$ | |
| $C := f_5(C)$ | $C := f_5(C)$ | | |
| **Write**$(C)$ | **Write**$(C)$ | | |
| $B := f_6(B)$ | | | $B := f_6(B)$ |
| **Write**$(B)$ | | | **Write**$(B)$ |

Time ↓



# Testing of Serializability



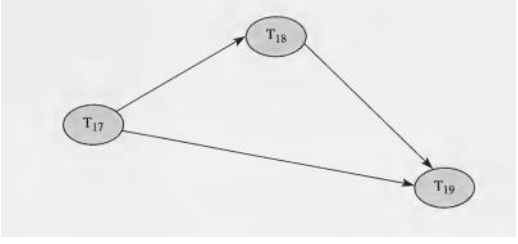
## Testing of Serializability

| Schedule | Transaction $T_{17}$ | Transaction $T_{18}$ | Transaction $T_{19}$ |
|---|---|---|---|
| **Read**$(A)$ | **Read**$(A)$ | | |
| $A := f_1(A)$ | $A := f_1(A)$ | | |
| **Read**$(C)$ | **Read**$(C)$ | | |
| **Write**$(A)$ | **Write**$(A)$ | | |
| $A := f_2(C)$ | $A := f_2(C)$ | | |
| **Read**$(B)$ | | **Read**$(B)$ | |
| **Write**$(C)$ | **Write**$(C)$ | | |
| **Read**$(A)$ | | **Read**$(A)$ | |
| **Read**$(C)$ | | | **Read**$(C)$ |
| $B := f_3(B)$ | | $B := f_3(B)$ | |
| **Write**$(B)$ | | **Write**$(B)$ | |
| $C := f_4(C)$ | | | $C := f_4(C)$ |
| **Read**$(B)$ | | | **Read**$(B)$ |
| **Write**$(C)$ | | | **Write**$(C)$ |
| $A := f_5(A)$ | | $A := f_5(A)$ | |
| **Write**$(A)$ | | **Write**$(A)$ | |
| $B := f_6(B)$ | | | $B := f_6(B)$ |
| **Write**$(B)$ | | | **Write**$(B)$ |

Time ↓



## Testing of Serializability





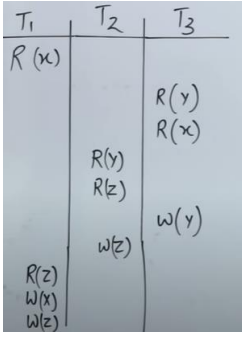

## Conflict Serializable

- R-W
- W-R
- W-W
- Check for loop or cycle,
- If not present then,

its conflict serializable

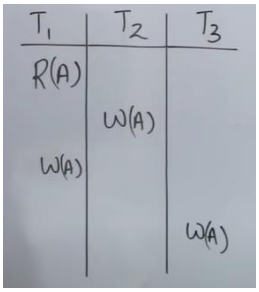
Thus, can be converted to serializable and will be consistent.

Find indegree=0.



## View Serializable

- Check for loop, if found it is not conflict, it's a view serializable.






## Concurrency Control

- If all schedules in a concurrent environment are restricted to serializable schedules, the result obtained will be consistent with some serial execution of the transactions and will be considered correct.
- However, using only serial schedules unnecessarily limits the degree of concurrency. Furthermore, testing for serializability of a schedule is not only computationally expensive but it is an after-the-fact technique and impractical.



## Concurrency Control

- Locking
  - Exclusive locks
  - Shared locks

Current state of locking of data-item

| Lock mode of request | Unlocked | Shared | Exclusive |
|---|---|---|---|
| Unlock | - | yes | yes |
| Shared | yes | yes | no |
| Exclusive | yes | no | no |

- Timestamp-based order
- Optimistic Scheduling
- Multiversion Technique

## Concurrency Control (Locking)

| Transaction $T_5$ | Transaction $T_6$ |
|---|---|
| **Lockx**(*A*) | **Lockx**(*Sum*) |
| **Read**(*A*) | *Sum* := 0 |
| *A* := *A* − 100 | **Locks**(*A*) |
| **Write**(*A*) | **Read**(*A*) |
| **Unlock**(*A*) | *Sum* := *Sum* + *A* |
| **Lockx**(*B*) | **Unlock**(*A*) |
| **Read**(*B*) | **Locks**(*B*) |
| *B* := *B* + 100 | **Read**(*B*) |
| **Write**(*B*) | *Sum* := *Sum* + *B* |
| **Unlock**(*B*) | **Write**(*Sum*) |
| | **Unlock**(*B*) |
| | **Unlock**(*Sum*) |



## Concurrency Control (Inconsistent read)

| Schedule | Transaction $T_5$ | Transaction $T_6$ |
|---|---|---|
| **Lockx**(*Sum*) | | |
| *Sum* := 0 | | |
| **Locks**(*A*) | | |
| **Read**(*A*) | | |
| *Sum* := *Sum* + *A* | | |
| **Unlock**(*A*) | | |
| **Lockx**(*A*) | **Lockx**(*A*) | |
| **Read**(*A*) | **Read**(*A*) | |
| *A* := *A* − 100 | *A* := *A* − 100 | |
| **Write**(*A*) | **Write**(*A*) | |
| **Unlock**(*A*) | **Unlock**(*A*) | |
| **Lockx**(*B*) | **Lockx**(*B*) | |
| **Read**(*B*) | **Read**(*B*) | |
| *B* := *B* + 100 | *B* := *B* + 100 | |
| **Write**(*B*) | **Write**(*B*) | |
| **Unlock**(*B*) | **Unlock**(*B*) | |
| **Locks**(*B*) | | **Locks**(*B*) |
| **Read**(*B*) | | **Read**(*B*) |
| *Sum* := *Sum* + *B* | | *Sum* := *Sum* + *B* |
| **Write**(*Sum*) | | **Write**(*Sum*) |
| **Unlock**(*B*) | | **Unlock**(*B*) |
| **Unlock**(*Sum*) | | **Unlock**(*Sum*) |

Time ↓



## Concurrency Control (solution for inconsistent read)

| Schedule | Transaction $T_{20}$ | Transaction $T_{21}$ |
|---|---|---|
| **Lockx**(*Sum*) | | |
| *Sum* := 0 | | |
| **Locks**(*A*) | | |
| **Read**(*A*) | | |
| *Sum* := *Sum* + *A* | | |
| **Locks**(*B*) | | |
| **Read**(*B*) | | |
| *Sum* := *Sum* + *B* | | |
| **Write**(*Sum*) | | |
| **Unlock**(*B*) | | |
| **Unlock**(*A*) | | |
| **Unlock**(*Sum*) | | |
| **Lockx**(*A*) | **Lockx**(*A*) | |
| **Read**(*A*) | **Read**(*A*) | |
| *A* := *A* − 100 | *A* := *A* − 100 | |
| **Write**(*A*) | **Write**(*A*) | |
| **Lockx**(*B*) | **Lockx**(*B*) | |
| **Unlock**(*A*) | **Unlock**(*A*) | |
| **Read**(*B*) | **Read**(*B*) | |
| *B* := *B* + 100 | *B* := *B* + 100 | |
| **Write**(*B*) | **Write**(*B*) | |
| **Unlock**(*B*) | **Unlock**(*B*) | |

Time ↓

## Concurrency Control (Two phase locking)

- The transaction must involve locking and unlocking as monotonic.
- All locks are first acquired before any of the locks are released.
- Once a lock is released, no additional locks are requested.
- The release of the locks is delayed until all locks on all data-items required by the transaction have been acquired.
- Growing phase and Contracting phase.



## Concurrency Control (Granularity of Locking)

- So far we have assumed that a data-item can be **locked.** However, we have not defined explicitly what the data-item is.
- If the size or granularity of the data-item is very large, for instance the entire database, then of course the overhead of locking is very small.
- If the granularity of the data-item is very small (for example, a data- item could be the field of a record), then the degree of concurrency can be fairly high, although the overhead of locking in this case can be considerable.



## Concurrency Control (Time Stamp based order)

- In the timestamp-based method, a serial order is created among the concurrent transaction by assigning to each transaction a unique nondecreasing number. The usual value assigned to each transaction is the system clock value at the start of the transaction, hence the name **timestamp ordering.**
- The serializability that the system enforces is the chronological order of the timestamps of the concurrent transactions. If two transaction Tj and Tj with the time stamp values tj and tj respectively, such that tj < tj, are to run concurrently, then the schedule produced by the system is equivalent to running the older transaction Tj first, followed by the younger one, Tj.

## Concurrency Control (Time Stamp based order)

- A conflict is said to occur when an older transaction tries to read a value that is written by a younger transaction or when an older transaction tries to modify a value already read or writ- \ ten by a younger transaction.
- Data-item X is thus represented by a triple X: {x, Wx, Rx where each component of the triple is interpreted as given below:
- x, the value of the data-item X
- Wx, the write timestamp value, the largest timestamp value of any transaction that was allowed to write a value of X.
- Rx, the read timestamp value, the largest timestamp value of any transaction that was allowed to read the current value X.



## Concurrency Control (Optimistic Scheduling)

- In the optimistic scheduling scheme, the philosophy is to assume that all data-items can be successfully updated at the end of a transaction and to read in the values for data- items without any locking. Reading is done when required and if any data-item is found to be inconsistent (with respect to the value read in) at the end of a transaction, then the transaction is rolled back.
- It has following three phases:
  - Read
  - Validation
  - Write



## Concurrency Control (Multiversion Technique)

- In a database system that uses the Multiversion concurrency scheme, each write of a data-item, e.g., X, is achieved by making a new copy or version (hence the name **Multiversion**) of data-item X. The Multiversion scheme, which is also called a time domain addressing scheme, follows the accounting principle of never over writing a transaction.
- Any changes are achieved by entering compensating transactions. In this way, a history of the evolution of the value of a data-item is recorded in the database.

## Deadlock and Starvation

- Suppose we have a situation where transaction Ta has claimed data-item ra and is waiting for data-item rb. Data-item rb, however, has been claimed by transaction Tb, which in turn is waiting for data-item rc. This chain of transactions holding some data-items and waiting for additional data-items continues until we come to trans¬ action T„ which has claimed data-item r, and is waiting for data-item ra. We know that data-item ra is held by transaction Ta!
- If none of these transactions is willing to release the data-items they are holding, none of these transactions can proceed. This is deadlock.

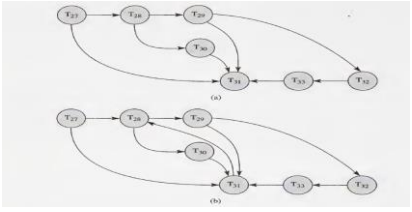


## Deadlock Detection

In order for the system to detect a deadlock, it must have the following information:

- The current set of transactions
- The current allocations of data-items to each of the transactions
- The current set of data-items for which each of the transactions is waiting.

The algorithm to detect a deadlock is based on the detection of such a circular chain in the current system wait-for graph.





## Deadlock Recovery

- To recover from deadlock, the cycles in the wait-for graph must be broken.
- The transaction must be roll back until the system exhibits no further deadlock situation.
- Which transaction must be roll back depends upon:
  - The progress of the transaction and the number of data-items it has used and modified. It is preferable to roll back a transaction that has just started or has not modified any data-item, rather than one that has run for a considerable time and/or has modified many data-items.
  - It is preferable not to roll back a transaction if it has almost run to completion and/or it needs very few additional data-items before its termination.

## Deadlock Avoidance

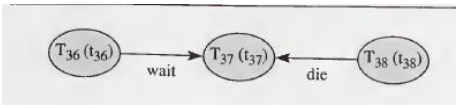
- In the deadlock avoidance scheme, care is taken to ensure that a circular chain of processes holding some resources and waiting for additional ones held by other trans¬ actions in the chain never occurs.
- One of the simplest methods of avoiding a deadlock situation is to lock all data- items at the beginning of a transaction.
- Another approach used in avoiding deadlock is assigning an order to the data- items and requiring the transactions to request locks in a given order, such as only ascending order.



## Deadlock Avoidance

Wait-die

- If the requesting transaction is older than the transaction that holds the lock on the requested data-item, the requesting transaction is allowed to wait.
- If the requesting transaction is younger than the transaction that holds the lock on the requested data-item, the requesting transaction is aborted and rolled back.






## Deadlock Avoidance

Wound-wait

- An opposite approach to the wait-die scheme is called the wound-wait scheme. Here the decision whether to wait or abort is as follows:
- If a younger transaction holds a data-item requested by an older one, the younger transaction is the one that would be aborted and rolled back (the younger transaction is wounded by the older transaction and dies!).
- If a younger transaction requests a data-item held by an older transaction, the younger transaction is allowed to wait.


## Time Stamp Ordering (Question)

Rules:

1.Transaction Ti issues a read operation Read (A)

If WTS (A) >TS (Ti), Rollback Ti.
Otherwise, Execute R(A) operation
Set RTS (A) =Max {RTS(A), Ts(Ti)}

2.Transaction Ti issues a Write operation Write (A)

If RTS(A)>TS(Ti), Rollback Ti.
If WTS(A)>TS(Ti), Rollback Ti
Otherwise, execute Write (A) operation
Set WTS (A)=Ts(Ti)



## Time Stamp Ordering (Question)

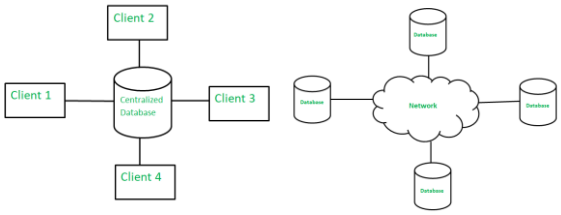
| T1 | T2 | T3 |
|---|---|---|
| R(A) | | |
| | R(B) | |
| W(C) | | |
| | | R(B) |
| R(C) | | |
| | W(B) | |
| | | W(A) |



## Concepts of OODBMS

- An object-oriented database management system (OODBMS), sometimes shortened to ODBMS for object database management system, is a database management system (DBMS) that supports the modeling and creation of data as objects.
- Data Encapsulation
- Polymorphism
- Performance
- Some popular examples of OODBMSs include MongoDB, Apache Cassandra, and ObjectDB.

## Distributed Data Base Management Systems






## Overview of Database Security Concepts

- Data base security: The mechanisms protect the database against intentional or accidental threats.
- Three are 3 main aspects
  - Confidentiality
  - Integrity
  - Availability




## Confidentiality

- It is protecting the database from unauthorized users.
- Ensures that users are allowed to do the things they are trying to do.
- Encryption is a technique or a process by which the data is encoded in such a way that only authorized users are able to read the data.

## Integrity

- Data integrity is the overall accuracy, completeness, and consistency of data.
- It is maintained by a collection of processes, rules, and standards implemented during the design phase.
- Back up data. Backup copies of data are essential in the event that data is lost or corrupted.
- Verify and validate data.



## Availability

- Database Availability means the period of time when the database engines are functional and the database engines' processes are executing and allowing end-users, whose connections reach the server, to access the database through their usual logon procedures.
- Availability concerns both the accessibility and continuity of information. Data that is not accessible quickly can prevent the delivery of services, costing an organization time and revenue.