**UNIT-IV: Transaction Processing Concepts**

**Transaction processing and Concurrency Control:** Definition of Transaction, Desirable ACID properties, overview of serializability, serializable and non serializable transactions

**Concurrency Control:** Definition of concurrency, lost update, dirty read and incorrect summary problems due to concurrency

**Concurrency Control Techniques:** Overview of Locking, 2PL, Timestamp ordering, multiversioning, validation.

**Elementary concepts of Database security:** System failure, Backup and Recovery Techniques, Authorization and authentication

**T**he concept of transaction provides a mechanism for describing logical units of database processing. **Transaction processing systems** are systems with large databases and hundreds of concurrent users that are executing database transactions. Examples of such systems include ***systems for reservations, banking, credit card processing, stock markets, supermarket checkout,*** and other similar systems. They require high availability and fast response time for hundreds of concurrent users.

The main concepts that are needed in transaction processing systems is ***transaction***, which is used to represent a logical unit of database processing that must be completed in its entirety to ensure correctness.

The concurrency control problem occurs when multiple transactions submitted by various users interfere with one another in a way that produces incorrect results.

**Introduction to Transaction Processing**

**Single-User Versus Multiuser Systems**

A DBMS is **single-user** if at most one user at a time can use the system, and it is **multiuser** if many users can use the system—and hence access the database—concurrently.

* + Most DBMS are **multiuser** (e.g., airline reservation system).
  + ***Multiprogramming operating systems***allow the computer to execute multiple programs (or processes) at the same time (having one CPU, concurrent execution of processes is actually interleaved).
  + If the computer has multiple hardware processors (CPUs), ***parallel processing***of multiple processes is possible.

**Introduction to Transactions, Read and Write Operations & DBMS Buffers**

A **Transaction** is a logical unit of database processing that includes one or more access operations ((e.g., insertion, deletion, modification, or retrieval operations).

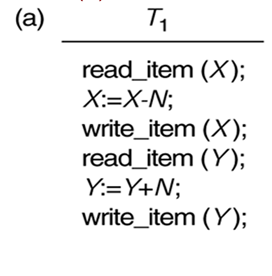
The database operations that form a transaction can either be embedded within an application program or they can be specified interactively via a high-level query language such as SQL. One way of specifying the transaction boundaries is by specifying explicit **begin transaction** and **end transaction** statements in an application program; in this case, all database access operations between the two are considered as forming one transaction.

A single application program may contain more than one transaction if it contains several transaction boundaries. If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a **read-only transaction.**

**SIMPLE MODEL OF A DATABASE** (for purposes of discussing transactions):

* **A database** is a collection of named data items.
* **Granularity** The size of a data item is called its **granularity,** and it can be a field of some record in the database, or it may be a larger unit such as a record or even a whole disk block
* Basic operations are **read** and **write**
  + **read\_item(X**): Reads a database item named X into a program variable. To simplify our notation, we assume that the program variable is also named X.
  + **write\_item(X**): Writes the value of program variable X into the database item named X.
* **read\_item(X)** command includes the following steps:
  + Find the address of the **disk block** that contains item X.
  + Copy that disk block into a buffer in **main memory** (if that disk block is not already in some main memory buffer).
  + Copy item X from the buffer to the **program variable** named X.
* **write\_item(X**) command includes the following steps:
  + Find the address of the **disk block** that contains item X.
  + Copy that disk block into a buffer in **main memory** (if that disk block is not already in some main memory buffer).
  + Copy item X from the **program variable** named X into its correct location in the buffer.
  + Store the updated block from the buffer back to **disk** (either immediately or at some later point in time).

Example: Let T1 be a transaction that transfer N=50 from Account X to Account Y. This can be defined in fig(a) as:



**Desirable Properties of Transactions (ACID properties)**

To ensure integrity of data, we require that the database system maintains the following properties of the transactions. These are often called the **ACID properties**. The following are the ACID properties:

1. **Atomicity:** A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.

2. **Consistency preservation:** A transaction is consistency preserving if its complete execution take(s) the database from one consistent state to another.

3. **Isolation:** A transaction should appear as though it is being executed in isolation from other transactions. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.

4. **Durability or permanency:** The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

**Transaction States**

A transaction is an atomic unit of work that is either completed in its entirety or not done at all. For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts (see below). Hence, the recovery manager keeps track of the following operations:

* **Active**, the initial state; the transaction stays in this state while it is executing.
* **Partially committed**, after the final statement has been executed
* **Failed**, after the discovery that normal execution can no longer proceed.
* **Aborted**, after the transaction has been rolled backed and the database has been restored to its state prior to the start of transaction.
* **Committed**, after successful completion.

A transaction must be in one of these states.

Begin

Transaction

Read

Write

Abort

Abort

Commit

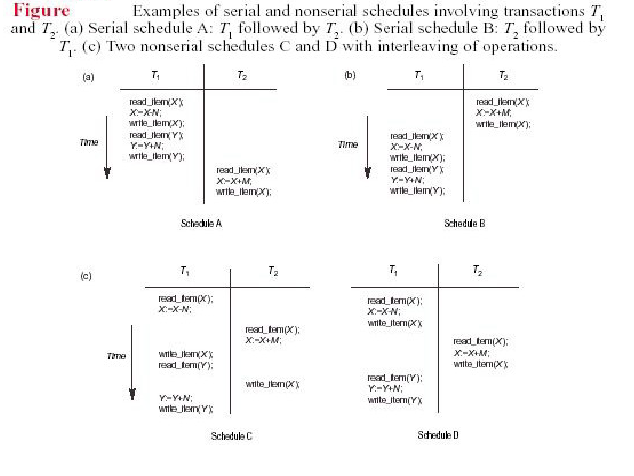
**Overview of Schedule, Serializability, Serializable and Non-serializable transactions**

**Schedules (Histories) of Transactions**

When transactions are executing concurrently in an interleaved fashion, then the order of execution of operations from the various transactions is known as a **schedule** (or **history)**.

A **schedule** (or **history**) *S* of *n* transactions T1,T2 , ...,Tn is an ordering of the operations of the transactions subject to the constraint that, for each transaction Ti, that participates in *S,* the operations of in *S* must appear in the same order in which they occur in.

For the purpose of recovery and concurrency control, we are mainly interested in the read\_item and write\_item operations of the transactions, as well as the commit and abort operations. A shorthand notation for describing a schedule uses the symbols *r, w, c,* and *a* for the operations read\_item, write\_item, commit, and abort, respectively.



For example, the schedule of Figure (a),(b),(c),(d) shown above which we shall call **Sa Sb Sc Sd** , can be written as follows in this notation:

* **Sa : r1(X); w1(X); r1(Y); w1(Y); r2(X);w2(X);**
* **Sb: r2(X); w2(X); r1(X); w1(X); r1(Y);w1(Y);**
* **Sc : r1(X); r2(X); w1(X); r1(Y); w2(X); w1(Y);**
* **Sd: r1(X); w1(X); r2(X); w2(X); r1(Y); a1;**

Two operations in a schedule are said to **conflict** if they satisfy all three of the following conditions:

* + they belong to different transactions;
  + they access the same item *X;* and
  + at least one of the operations is a write\_item(*X*)*.*

For example, in schedule **Sc**, the operations r1(*X*) and w2(*X*) conflict, as do the operations r2(*X*) and w1(*X*), and the operations w1(*X*) and w2(*X*).

However, the operations r1(*X*) and r2(*X*) do not conflict, since they are both read operations; the operations w2(*X*) and w1(*Y*) do not conflict, because they operate on distinct data items *X* and *Y;* and the operations r1(*X*) and w1(*X*) do not conflict, because they belong to the same transaction.

**Serializability of Schedules**

* The concept of **serializability of schedules** is used to identify which schedules are correct when transaction executions have interleaving of their operations in the schedules.

Two transactions T1 & T2 execute serially in a schedule S if it :

i) Execute all the operations of transaction T1 (in sequence) followed by all the

operations of transaction T2(in sequence).

ii) Execute all the operations of transaction T2 (in sequence) followed by all the

operations of transaction T1 (in sequence).

Otherwise, they execute concurrently.

* A schedule S is ***serial*** if for any transaction Ti executing in S, all the operations in Ti are executed consecutively in S; otherwise, S is called ***nonserial***. In other words, only one transaction at a time is executed in S. There is no interleaving. e.g of serial schedule:

i) T1 is followed by T2 & ii) T2 is followed by T1.

* A schedule *S* of *n* transactions is ***serializable*** if it is equivalent to some serial schedule of the same *n* transactions.

A schedule *S* of *n* transactions T1, T2, ..., Tn, is said to be a **complete schedule** if the following conditions hold:

* The operations in *S* are exactly those operations in T1, T2, ..., Tn, including a commit or abort operation as the last operation for each transaction in the schedule.
* For any pair of operations from the same transaction Ti, their order of appearance in *S* is the same as their order of appearance in Ti.
* For any two conflicting operations, one of the two must occur before the other in the schedule.

**CONCURRENCY CONTROL**

**Concurrency in a DBMS**

Concurrent execution of user programs is essential for good DBMS performance. Because disk accesses are frequent, and relatively slow, it is important to keep the CPU busy all the time by working on several user programs concurrently. Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions. Each transaction must leave the database in a consistent state.

**Advantages:**

**-** increase the throughput of the system.

- minimize response/waiting time for each transaction**.**

**Why Concurrency Control Is Needed**

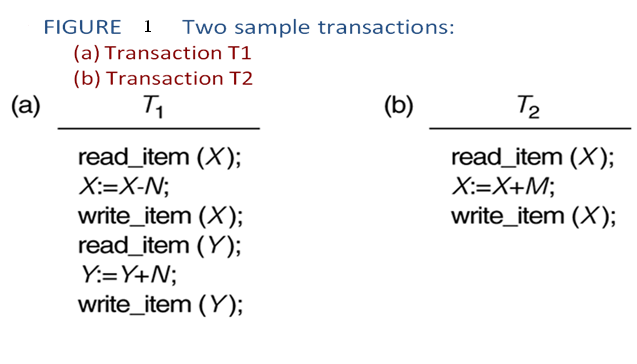
Concurrency control is mainly concerned with the database access commands in a transaction. Transactions submitted by the various users may execute concurrently and may access and update the same database items. If this concurrent execution is uncontrolled, it may lead to problems, such as an inconsistent database.

Several problems can occur when concurrent transactions execute in an uncontrolled manner.

Some of these problems are:

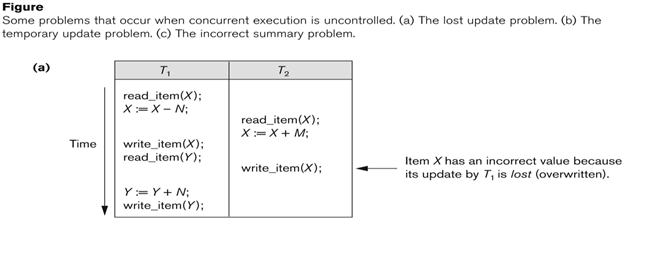
1. Lost Update Problem
2. Dirty Read Problem &
3. The Incorrect Summary Problem

Consider an example by referring figure 1 a & b, where T1 is a transaction that transfer N=50 from Account X to Account Y. and T2 is a simple transaction that simply add M to Account X referenced in T1..



**i) The Lost Update Problem:** This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the **value of some database item incorrect**.

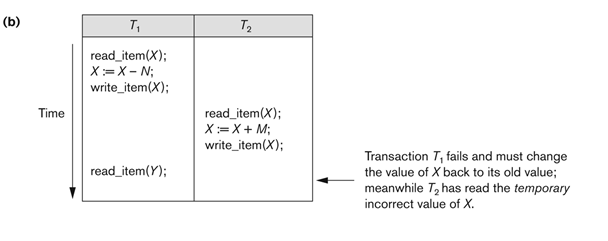
Suppose that transactions are submitted at approximately the same time, and suppose that their operations are interleaved as shown in Figure (a) below; then the final value of item *X* is incorrect, because reads the value of *X before* changes it in the database, and hence the updated value resulting from is lost. For example, At Initial, if *X* = 80, *N* = 5 and *M* = 4, then the final result should be *X* = 79; but in the interleaving of operations shown in Figure (a), it is *X* = 84 because the update in that removed 5 from *X* was *lost.*

****

**ii) The Temporary Update (or Dirty Read) Problem**

* + This problem occurs when one transaction updates a database item and then the **transaction fails** for some reason.
  + The updated item is **accessed by another transaction** before it is changed back to its original value.

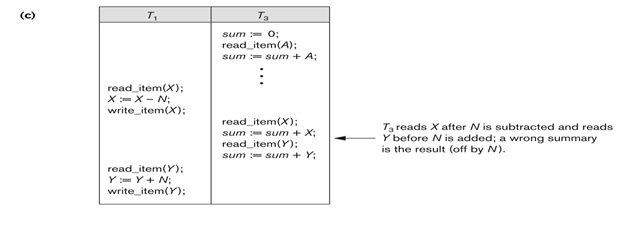
Figure (b) below shows an example where updates item *X* and then fails before completion, so the system must change *X* back to its original value. Before it can do so, however, transaction reads the "temporary" value of *X,* which will not be recorded permanently in the database because of the failure. The value of item *X* that is read by is called ***dirty data****,* because it has been created by a transaction that has not completed and committed yet; hence, this problem is also known as the *dirty read problem.*

****

**iii) The Incorrect Summary Problem**

If one transaction is calculating an **aggregate summary function** on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.

For example, suppose that a transaction is calculating the total number of amounts on all the accounts; meanwhile, transaction is executing. If the interleaving of operations shown in Figure (c) occurs, the result of will be off by an amount *N* because reads the value of *X after N* have been subtracted from it but reads the value of *Y before* those *N*  have been added to it.



**Concurrency Control Techniques**

There are number of concurrency control techniques that are used to ensure the isolation property of concurrently executing transactions. Most of these techniques ensure serializability of schedules using **protocols** (sets of rules) that guarantee serializability. One important set of protocols employs the technique of **locking** data items to prevent multiple transactions from accessing the items concurrently. Locking protocols are used in most commercial DBMSs.

Another set of concurrency control protocols use **timestamps.** A timestamp is a unique identifier for each transaction, generated by the system.

The **multiversion** concurrency control protocols use multiple versions of a data item.

A protocol based on the concept of **validation** or **certification** of a transaction after it executes its operations; is sometimes called **optimistic protocol.**

1. **Locking Techniques for Concurrency Control**

Transaction processing system usually allows multiple transactions to run concurrently. By allowing multiple transactions to run concurrently will improve the performance of the system in terms of increased throughout or improved response time, but this allows causes several complications with consistency of the data. Ensuring consistency in spite of concurrent execution of transaction require extra work, which is performed by the concurrency controller system of DBMS.

## What is Lock?

A lock is a variable associated with a data item that describes the status of the item withrespect to possible operations that can be applied to it. Generally, there is one lock for each dataitem in the database. Locks are used as a means of synchronizing the access by concurrenttransactions to the database item.

**Types of Locks**

Generally 2 types of locks are used in concurrency control.

1. B[**inary**](http://ecomputernotes.com/database-system/rdbms/type-of-lock-in-dbms) **locks**, which are simple but restrictive and so are not used in practice.
2. **Shared/Exclusive locks**, which provide more general locking capabilities and are used in practical database locking schemes.

**i) Binary Locks**

A [binary](http://ecomputernotes.com/database-system/rdbms/type-of-lock-in-dbms) lock can have two states or values: locked and unlocked. A distinct lock is associated with each database item *A.* If the value of the lock on *A* is 1, item *A* cannot be accessed by a database operation that requests the item. If the value of the lock on *A* is 0 then item can be accessed when requested.

We refer to the current value of the lock associated with item *A* as *LOCK (A).*

There are two operations, lock\_item(A) and unlock(A) that are used with [**binary**](http://ecomputernotes.com/database-system/rdbms/type-of-lock-in-dbms) locking. A transaction requests access to an item *A* by first issuing a **lock\_*item(A)***operation.

* If LOCK (A) = 1, the transaction is forced to wait.
* If LOCK (A) = 0 it is set to 1 (the transaction locks the item) and the transaction is allowed to access item *A.*

When the transaction is through using the item, it issues an **unlock*(A)***operation, which sets *LOCK (A)* to 0 (unlocks the item) so that *A* may be accessed by other transactions. Hence binary lock enforces mutual exclusion on the data item.

**Rules of Binary Locks**

If the simple binary locking scheme described here is used, every transaction must obey the following rules:

1. A transaction must issue the operation lock\_item(A) before any read\_item (A) or write, item operations are performed in T.
2. A transaction T must issue the operation unlock(A) after all read\_item (A) and

write\_item (A) operations are completed in T.

1. A transaction T will not issue a lock *\_item (A)* operation if it already holds the lock on Item *A.*
2. A transaction T will not issue an unlock*(A)* operation unless it already holds the lock on item *A.*
3. The lock manager module of the DBMS can enforce these rules. Between the lock*\_item (A)* and unlock*(A)* operations in transaction T, is said to hold the lock on item *A.* At most one transaction can hold the lock on a particular item. Thus no two transactions can access the same item concurrently.

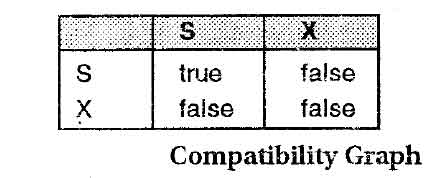
  Disadvantages of Binary Locks

As [binary](http://ecomputernotes.com/database-system/rdbms/type-of-lock-in-dbms) locking scheme is too restrictive for database items, because at most one transaction can hold a lock on a given item. So, binary locking system cannot be used for practical purpose.

**ii) Share/Exclusive (for Read/Write) Locks**

**Locking operations:** There are **three** locking operations called **read\_lock(A), write\_lock(A)** and **unlock(A)**. A lock associated with an item A, LOCK (A), now has three possible states: "read-locked", "write-locked," or "unlocked." A read-locked item is also called share-locked item because other transactions are allowed to read the item, whereas a write-locked item is caused exclusive-locked, because a single transaction exclusively holds the lock on the item.

**Compatibility of Locks** Suppose that there are A and B two different locking modes Read or Shared(S) & Write or Exclusive(X). If a transaction T1 requests a lock of mode on item Q on which transaction T2 currently hold a lock of mode B. If transaction can be granted lock, in spite of the presence of the mode B lock, then we say mode A is compatible with mode B. Such a function is shown in one matrix as shown below:

[](http://ecomputernotes.com/images/compatibility-graph.jpg)

The graphs shows that if two transactions only read the same data object they do not conf1ict, but if one transaction writes a data object and another either read or write the same data object, then they conflict with each other. A transaction requests a shared lock on data item Q by executing the read\_lock(Q) instruction. Similarly, an exclusive lock is requested through the write\_lock(Q) instruction. A data item Q can be unlocked via the unlock(Q) instruction.

**How Should Lock be Used?**

To access a data item, transaction T1 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, T1 is made to wait until all incompatible locks held by other transactions have been released

When we use the shared/exclusive locking scheme, the system must enforce the following rules:

* 1. A transaction T must issue the operation read\_lock(X) or write\_lock(X) before any read\_item(X) operation is performed in T.
  2. A transaction T must issue the operation write\_lock(X) before any write\_item(X) operation is performed in T.
  3. A transaction T must issue the operation unlock(X) after all read\_item(X) and write\_item(X) operations are completed in T.
  4. A transaction T will not issue a read\_lock(X) operation if it already holds a read (shared) lock or a write (exclusive) lock on item X. This rule may be relaxed.
  5. A transaction T will not issue a write\_lock(X) operation if it already holds a read (shared) lock or write (exclusive) lock on item X. This rule may be relaxed.
  6. A transaction T will not issue an unlock(X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.

For example, a transaction **T1’** after applying the read/write or shared/exclusive lock on **T1.**

**T1 T1’(equivalent transaction after applying read/write lock)**

|  |
| --- |
| **read(x);**  **read(y);**  **y=y+x;**  **write(y);** |

|  |
| --- |
| **read\_lock(x); Shared lock on x**  **read(x);**  **unlock(x); unlock x**  **write\_lock(y); exclusive lock on y**  **read(y);**  **y = y + x;**  **write(y);**  **unlock(y); unlock y** |

**Example:** Let x & y are two accounts that are accessed by transactions T1 & T2. T1 transfers 50/- from account x to y and is defined below.

Transaction **T1** **T2**: simply add x & y and display the sum

|  |
| --- |
| **read\_lock(y);**  **read(y);**  **unlock(y);**  **read\_lock(x);**  **read(x);**  **unlock(x);**  **sum=x+y;**  **display(sum);** |

|  |
| --- |
| **write\_lock(x);**  **read(x);**  **x = x – 50;**  **write(x);**  **unlock(x);**  **write\_lock(y);**  **read(y);**  **y = y + 50;**  **write(y);**  **unlock(y);** |

Suppose that the values of accounts x and y are Rs.200 and Rs.100, respectively. If these two transactions are executed serially, either in the order T1 and T2 or the order T2 and T1 then transaction T2 will display the value Rs.300. If however, these transactions are executed concurrently, as shown in Schedule 1, in this case, transaction T2 displays Rs.250, which is incorrect. The reason for this mistake is that the transaction TI unlocked data item **x** too early, as a result of which T2 shows an inconsistent state.

**T1                                    T2**

**write\_lock(x);**

**read(x);**

**x = x – 50;**

**write(x);**

**unlock(x);**

**read\_lock(y);**

**read(y);**

**unlock(y);**

**read\_lock(x);**

**read(x);**

**unlock(x);**

**sum=x+y;**

**display(sum);**

**write\_lock(y);**

**read(y);**

**y = y + 50;**

**write(y);**

**unlock(y);**

**Figure: Schedule 1**

Hence, it can be found that using read/write locking mechanism doesn’t guarantee serializibilty. Guaranteeing serializibilty can be done by the locking scheme called **2-phase locking** scheme or **2PL** mechanism.

**Two Phase Locking (2PL) Protocol**

The use of locks has helped us to create neat and clean concurrent schedule. The Two Phase Locking Protocol defines the rules of how to acquire the locks on a data item and how to release the locks. The Two Phase Locking Protocol assumes that a transaction can only be in one of two phases.

i) **Growing Phase:** During this phase, the transaction can only acquire locks but no existing lock can be released.

**ii)**  **Shrinking Phase:** During this phase, existing locks can be released but no new locks can be acquired.

The 2PC Locking protocol always results in a serializable schedule.

Example :: here the transaction T1 do not follow the 2PC Protocol, because the write\_lock(y) operation in T1 follows the unlock(x) operation. On enforcing the 2PC Locking scheme, the transaction T1 can be written as T1’ as follows:

|  |
| --- |
| **write\_lock(x);**  **read(x);**  **x = x – 50;**  **write(x);**  **unlock(x);**  **write\_lock(y);**  **read(y);**  **y = y + 50;**  **write(y);**  **unlock(y);** |

**T1 T1’**

|  |
| --- |
| **write\_lock(x);**  **read(x);**  **x = x – 50;**  **write(x);**  **write\_lock(y);**  **unlock(x);**  **read(y);**  **y = y + 50;**  **write(y);**  **unlock(y);** |

**Problems in Two-Phase Locking**

* **Deadlock -** It happens whenever a transaction waits for a lock to be unlock (to access the data).
* **Cascading roll back** -When the data requested by 1 transaction is held by some other transactions again and again and the requested data is not given.

**Concurrency Control Based on Timestamp Ordering**

**The Timestamp Ordering(TO) Algorithm**

The timestamp-ordering protocol ensures serializability among transaction in their conflicting read and write operations. This is the responsibility of the protocol system that the conflicting pair of tasks should be executed according to the timestamp values of the transactions.

**Timestamps**

A **timestamp** is a unique identifier created by the DBMS to identify a transaction. Typically, timestamp values are assigned in the order in which the transactions are submitted to the system, so a timestamp can be thought of as the ***transaction start time****.* The timestamp of transaction T can *be* referred as **TS(T).** Concurrency control techniques based on timestamp ordering do not use locks; hence, ***deadlocks cannot occur****.*

In this, a unique fixed timestamp is associated with each transaction to keep the order of the transaction. It is denoted by ***TS (Ti).***

**Example:** If a transaction T1 has been assigned timestamp TS (T1) and a new transaction TS (T2) enters the system, then **TS (T1) < TS (T2).**

Two methods for implementing TS

* Use the value of the system as the timestamp (System Clock).
* Use a logical counter that is incremented after a new timestamp has been assigned.

**Implementation Method**

The timestamp of a data item can be of the following two types:

* **WTS(Q) or W-timestamp (Q)**: This means the latest time when the data item Q has been written into.
* **RTS(Q) or R-timestamp (Q)**: This means the latest time when the data item Q has been read from.

These two timestamps are updated each time a successful read/write operation is performed on the data item Q.

Timestamp ordering protocol works as follows:

**1. If a transaction Ti issues read operation:**

If TS(Ti) < WTS(Q), then the read operation is rejected and the transaction Ti is restarted with a new timestamp value; otherwise, i.e if TS(Ti) >= WTS(Q) , the read operation is executed and RTS(Q) is set to max(RTS(Q), TS(Ti)).

**2.** **If a transaction Ti issues write operation:**

If TS (Ti) < RTS(Q) or TS (Ti) < WTS(Q), then the write operation is rejected or rollback and the transaction Ti is restarted with a new timestamp value; otherwise, if TS(Ti) >= RTS(Q) or TS(Ti) >= WTS(Q) , the write operation is executed and WTS(Q) is set to TS(Ti).

Example:

**Step T1 T2 T3 a b c**

**(200) (150) (175) RT(a)=0, RT(b)=0, RT(c)=0,**

**WT(a)=0 WT(b)=0 WT(c)=0**

1 Read(b) **RT(b)=200**

2 Read(a) **RT(a)=150**

3 Read(c) **RT(c)=175**

4 Write(b) **WT(b)=200**

5 Write(a) **WT(a)=200**

6 Write(c)

7 Write(a)

At step1, the operation read(b) executed, as

TS(T1) < WTS(b) i.e 200 < 0 does not follows , and

RTS(b) is set to max(RTS(b), TS(T1)) i.e RTS(b)=200

Similar rules for step 2, 3

For step 4 the operation write(b) executed, as

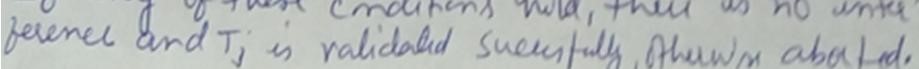
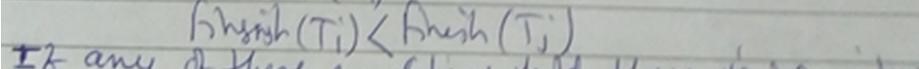
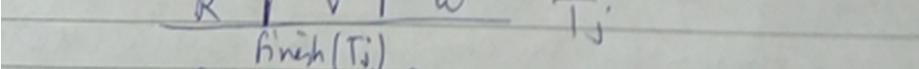
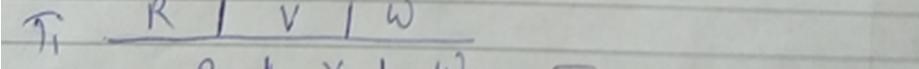
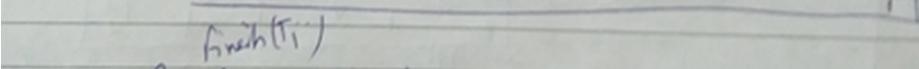
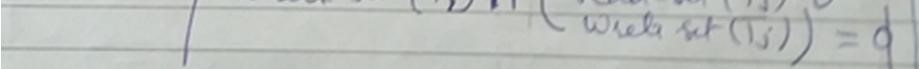
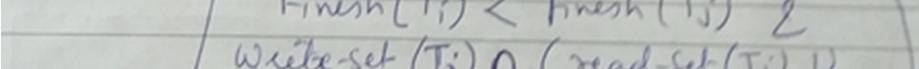
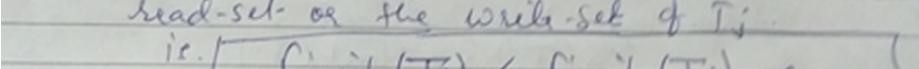
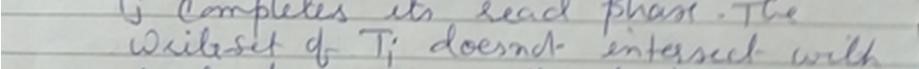
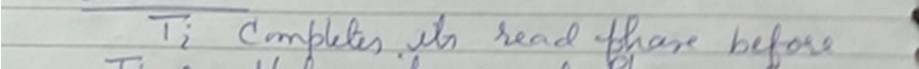
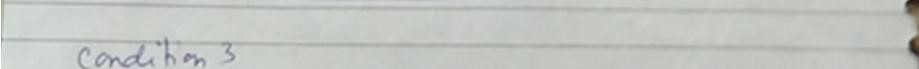
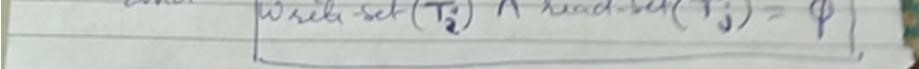
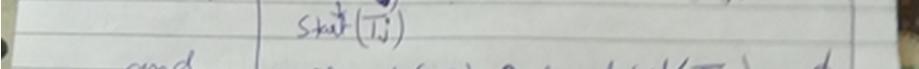
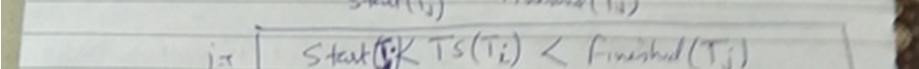
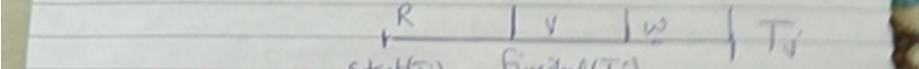
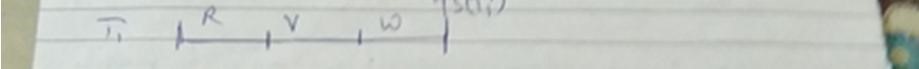
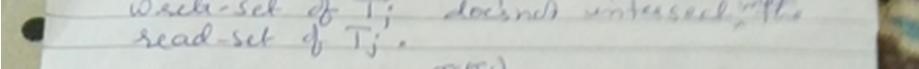
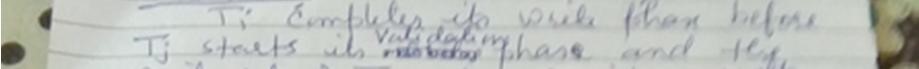
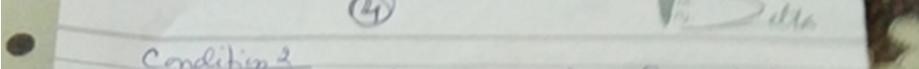
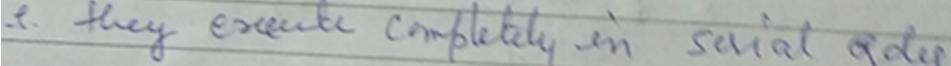
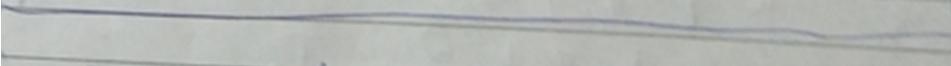
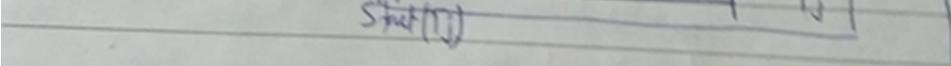
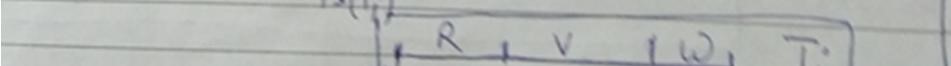
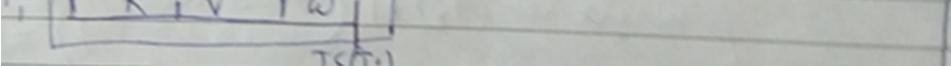
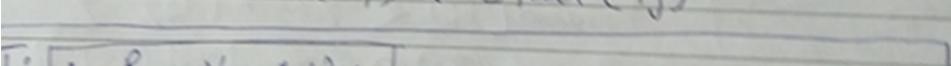
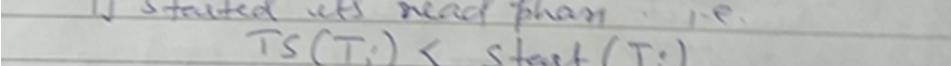
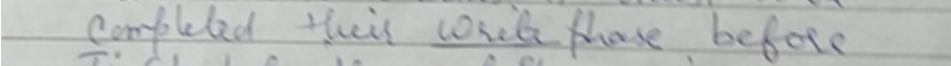
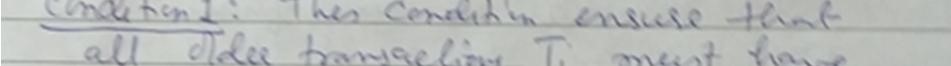
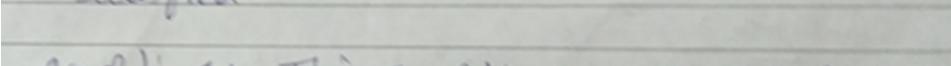
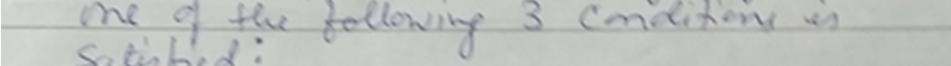
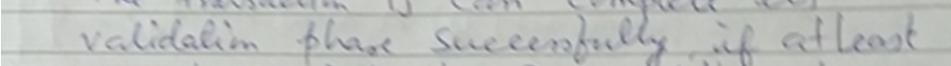
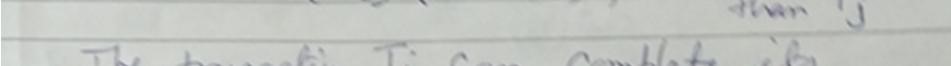
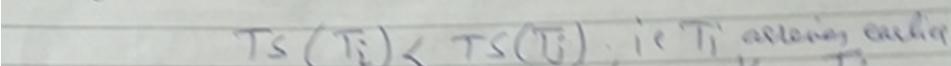
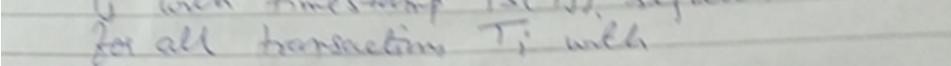
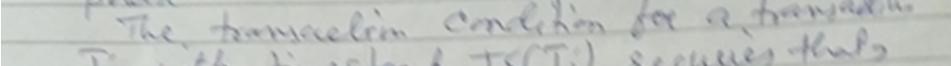
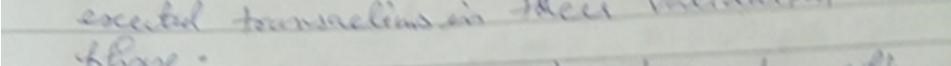
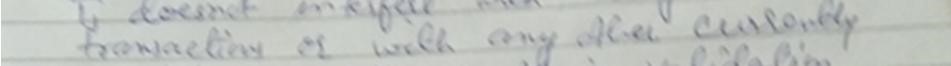
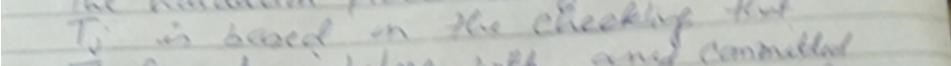
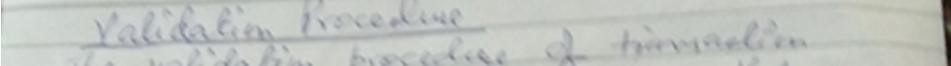
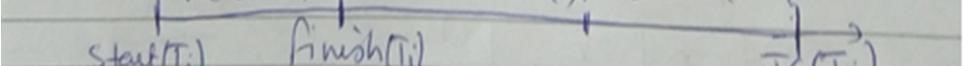
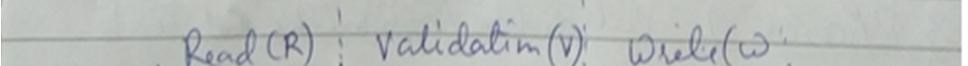
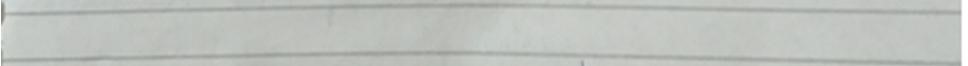
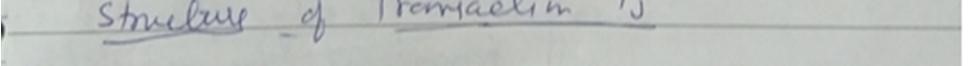
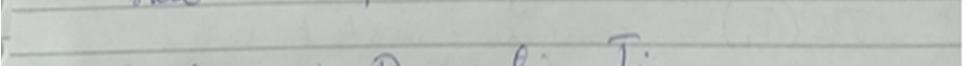
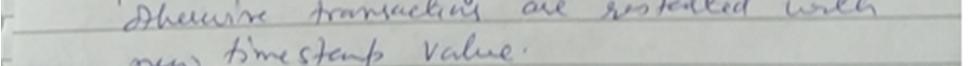
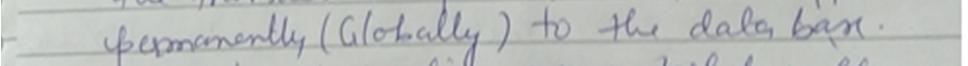
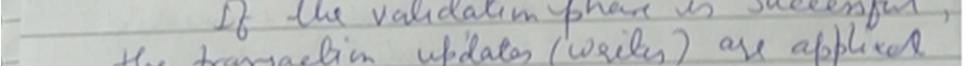
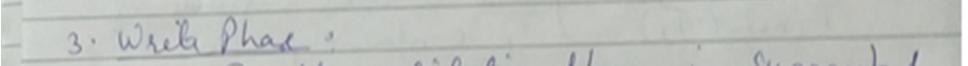
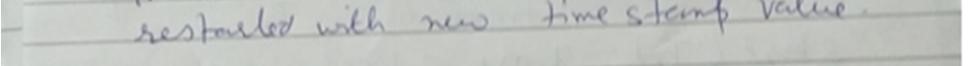
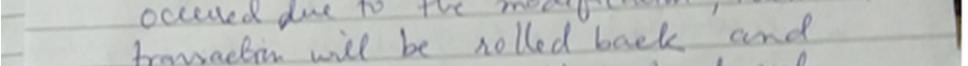
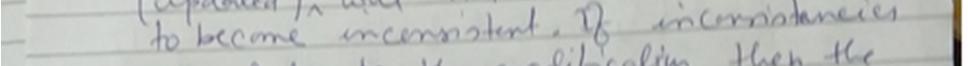
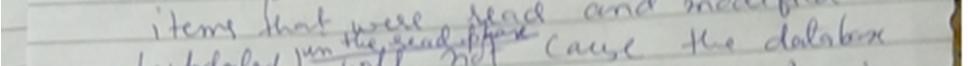
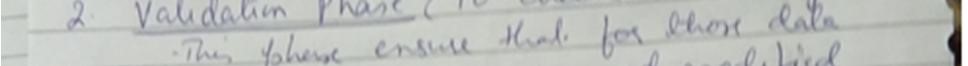
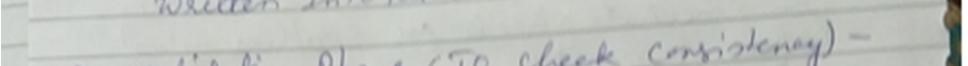
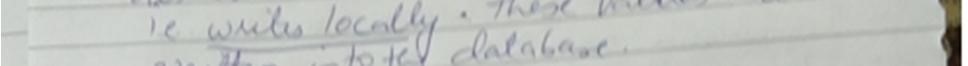
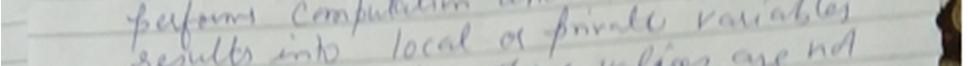
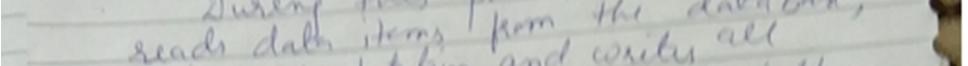
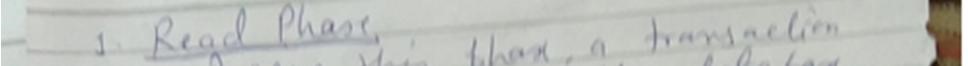
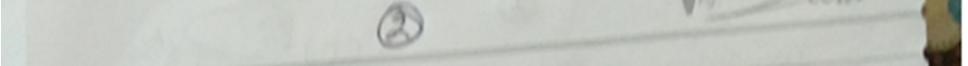
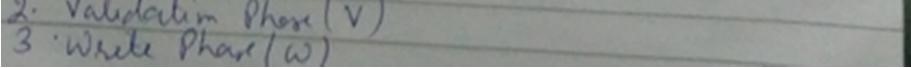
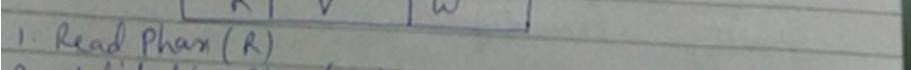
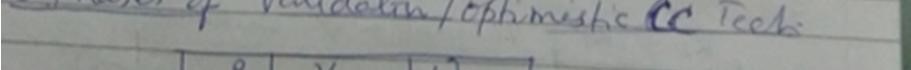
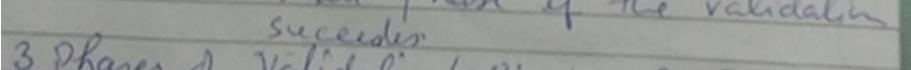
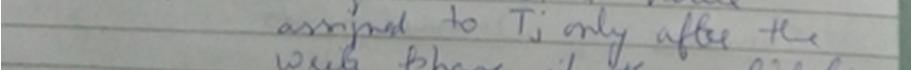
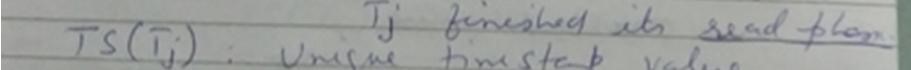
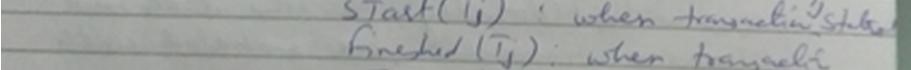
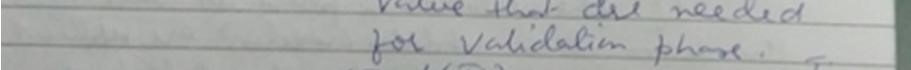
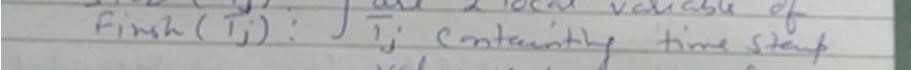
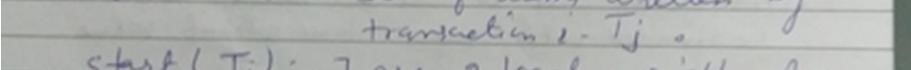
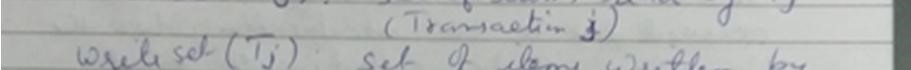
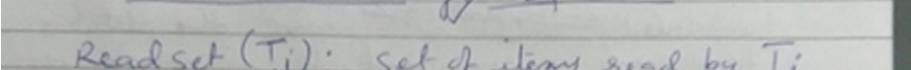
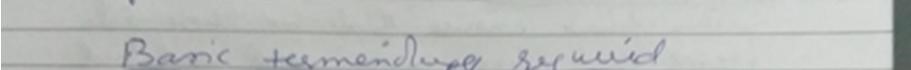
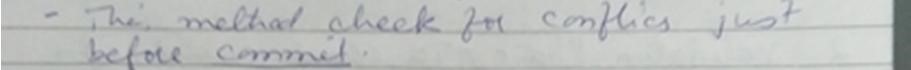
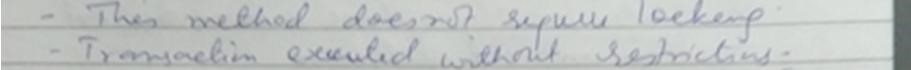
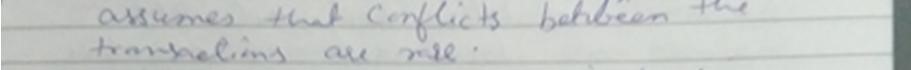
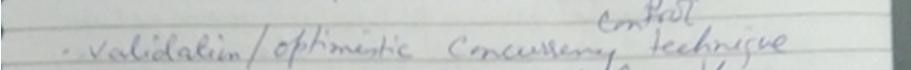
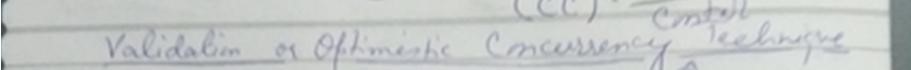
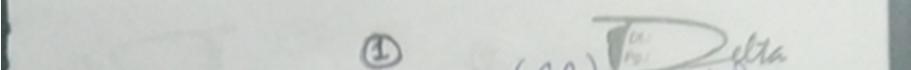
TS (T1) < RTS(b) or TS (T1) < WTS(b) i.e. 200 < 200 or 200 < 0 does not follows , and

WTS(b) is set to TS(T1) i.e WTS(b)=200

Step 5 the operation write(a)is also executed.

At step 6, however the operation write(c) is rejected as it does not follows the rule

At step 7, T3 tries to write a & the read value of a i.e RTS(a) =150. But TS(T3) > RT(a) , i.e 175 > 150, T3 need not to be aborted.



**Multiversion Concurrency Control Techniques**

**Q. What is multiversion concurrency control technique? Explain how multiversion concurrency control can be achieved by using Time Stamp Ordering.**

To improve database performance, multiversion concurrency control protocols are developed to extend the basic single version protocols. For single version databases, we have Two-phase Locking, Timestamp Ordering and Optimistic Concurrency Control. Consequently. For multiversion databases, there are Multiversion Two-phase Locking (MV2PL), Multiversion Timestamp Ordering (MVTSO), and Multiversion Optimistic Concurrency Control.

The basic idea behind multiversion concurrency control is to maintain one or more old versions (values) of data item when the item is updated.

When a transaction requires access to an item, an *appropriate* version is chosen to maintain the serializability of the currently executing schedule, if possible. The idea is that some read operations that would be rejected in other techniques can still be accepted by reading an *older version* of the item to maintain serializability. When a transaction writes an item, it writes a *new version* and the old version of the item is retained.

**Disadvantages:**

The drawback of multi version techniques is that more storage is needed to maintain multiple versions of the database items.

**Multiversion Technique Based on Timestamp Ordering**

In this method, several versions of each data item *X* are maintained. For *each version,* the value of version and the following two timestamps are kept:

1. **read\_TS:** The **read timestamp** of is the largest of all the timestamps of transactions that have successfully read version .

2. **write\_TS:** The **write timestamp** of is the timestamp of the transaction that wrote the value of version .

Whenever a transaction T is allowed to execute a write\_item(*X*) operation, a new version of item *X* is created, with both the write\_TS and the read\_TS set to TS(T). Correspondingly, when a transaction T is allowed to read the value of version *Xi*, the value of read\_TS() is set to max( read\_TS() , TS(T)).

To ensure serializability, the ***following two rules*** are used:

1. If transaction T issues a write\_item(*X*) operation, and version *i* of *X* has the highest write\_TS() of all versions of *X* that is also ***less than or equal to***TS(T), and read\_TS() > TS(T), then abort and roll back transaction T; otherwise, create a new version of *X* with read\_TS() = write\_TS() = TS(T).

2. If transaction T issues a read\_item(*X*) operation, find the version *i* of *X* that has the highest write\_TS() of all versions of *X* that is also ***less than or equal to***TS(T); then return the value of to transaction T, and set the value of read\_TS() to the larger of TS(T) and the current read\_TS().

As we can see in case 2, a read\_item(*X*) is always successful, since it finds the appropriate version to read based on the write\_TS of the various existing versions of *X*. In case 1, however, transaction T may be aborted and rolled back. This happens if T is attempting to write a version of *X* that should have been read by another transaction T whose timestamp is read\_TS(); however, T has already read version Xi, which was written by the transaction with timestamp equal to write\_TS(). If this conflict occurs, T is rolled back; otherwise, a new version of *X*, written by transaction T, is created.

**Introduction to Database Security**

**Types of Security issues:**

Database security is a broad area that addresses many issues, including the following:

* Various legal and ethical issues regarding the right to access certain information—
* Policy issues at the governmental, institutional, or corporate level as to what kinds of information should not be made publicly available.
* System-related issues such as the *system levels* at which various security functions should be enforced—for example, whether a security function should be handled at the physical hardware level, the operating system level, or the DBMS level.

**Database administrator**(**DBA) & Database Security**

The database administrator (DBA) is the central authority for managing a database system. The DBA’s responsibilities include granting privileges to users who need to use the system and classifying users and data in accordance with the policy of the organization. The DBA is responsible for the overall security of the database system. The following list of actions are used to control access to the DBMS:

1. **Account creation.** This action creates a new account and password for a user or a group of users to enable access to the DBMS.
2. **Privilege granting**. This action permits the DBA to grant certain privileges to certain accounts.
3. **Privilege revocation**. This action permits the DBA to revoke (cancel) certain privileges that were previously given to certain accounts.
4. **Security level assignment**. This action consists of assigning user accounts to the appropriate security clearance level.

**What is Database Backup?**

* Copying and archiving of computer data so it may be used to restore the original after a data loss event.
* Purpose is to recover data after it is lost from corruption or deletion.
* Second purpose is to recover data from an earlier time.

**What is** **Database recovery techniques**

Recovery techniques are used to ensure database consistency and transaction automaticity and durability despite failures.

**Why recovery is needed? (What causes a Transaction to fail?)**

There are several reasons that could cause a transaction to fail in the middle of execution

1. **A computer failure (system crash):** A hardware or software error occurs in the computer system during transaction execution. If the hardware crashes, the contents of the computer’s internal memory may be lost.
2. **A transaction or system error :** Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of a logical programming error. In addition, the user may interrupt the transaction during its execution.
3. **Local errors or exception conditions** detected by the transaction.
4. **Disk failure:** Some disk blocks may lose their data because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.
5. **Physical problems and catastrophes:** This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

There are many different approaches to recover a database:

**Manual reprocessing** & a variety of **automated recovery techniques**.

**1. Manual Reprocessing**

The database is periodically backed up (a database *save*) and all transactions applied since the last save are recorded. If the system crashes, the latest database save is restored and all of the transactions are re-applied (by users) to bring the database back up to the point just before the crash.

**Limitations:**

* Time required to reapply transactions
* Transactions might have other (physical) potential failures
* Reapplying concurrent transactions is not straight forward.

**2. Automated Recovery**

There are several types of automated recovery techniques including: D**eferred update, Immediate update, Shadow paging**, etc.

A transaction can be in one of the following states:

* **Active** - when the transaction just begins
* **Partially Committed** - after the last operation has completed (but before the commit point is reached)
* **Failed** - Normal operation is prevented (e.g., in the event of a system crash)
* **Aborted** - Transaction is rolled back. That is, all of its effects are undone
* **Committed** - Transaction completes all operations and moved the database to the next consistent state

**The System Log**

To be able to recover from failures that affect transactions, the system maintains a ***log***to keep

track of all transactions that affect the values of database items. Each transaction writes the following information to **the log**:

* **Start(T)** - the fact that transaction T has started
* **Write(T, X, old\_value, new\_value)** - the fact that transaction T has written to item X with the new\_value. old\_value is also maintained.
* **Read(T,X)-**the fact that transaction T has read data item X either:  
  **Commit(T)-**transaction T committed or **Abort(T)** - transaction T was aborted

**Recovery techniques use the following operations:**

**UNDO (Backward Recovery):** Similar to rollback except that it applies to a single operation rather than to a whole transaction.

**REDO (Forward Recovery):** This specifies that certain *transaction operations* must be *redone* to ensure that all the operations of a committed transaction have been applied successfully to the database.

**The 3 Important Recovery Techniques from transaction failures:**

**1. Deferred Update Recovery:** Also called ***NO-UNDO, REDO*** Technique

This technique defers or postpones any actual updates to the database until the transaction

reaches it commit point.

* During transaction execution, the updates are written to the log file.
* After the transaction reaches it commit point, the log file is force-written to disk, then the updates are recorded in the database.
* If the transaction fails before reaching its commit point, there is **no need to** ***UNDO*** any operations because the transaction has not affected the database on disk in any way.
* ***REDO*** may be necessary.

A typical deferred update protocol uses the **following rules**:

1. A transaction cannot change the database on disk until it reaches its commit point.
2. A transaction does not reach its commit point until all its update operations are

recorded in the log file and the log file is force-written to disk.

1. Because the database is never updated on disk until after the transactions commits,

there is never the need to UNDO any operations. That’s why a recovery techniques based on deferred update are therefore known as ***NO UNDO, REDO*** techniques.

**Example:**

**T1: Ra Rd Wd C**

**T2: Rb Wb Rd Wd C**

**T3: Ra Wa Rc Wc C**

**T4: Rb Wb Ra Wa C**

**Log file:**

Start(T1)

Write(T1, d, old, new)

Commit(T1)

Start(T4)

Write(T4, b, old, new)

Write(T4, a, old, new)

Commit(T4)

Start(T2)

Write(T2, b, old, new)

Start(T3)

Write(T3, a, old, new)

**s y s t e m c r a s h**

Since T1 and T4 committed, their changes were written to disk. However, T2 and T3 did not commit, hence their changes were not written to disk. To recover, we simply ignore those transactions that did not commit.

**Advantages:**

* Recovery is made easier.
* Any transaction that reached the commit point (from the log) has its writes applied to the database.
* All other transactions are ignored.
* Cascading rollback does not occur because no other transaction sees the work of another until it is committed.

**Disadvantages:**

Concurrency is limited: Must employ Strict 2PL which limits concurrency.

**2. Immediate Update Recovery: also called (UNDO, NO-REDO)**

* The database is physically updated before the transaction reaches to its commit point.
* UNDO may be necessary when a transaction fails.
* REDO is not needed.

A typical deferred update protocol uses the following **rules**:

1. In **immediate update** techniques, although the database can be updated immediately, update operations must be recorded in the log (on disk) before being applied to the database.
2. There is never a need to REDO any operations of committed transactions (this is called the **UNDO/NO-REDO recovery algorithm**).
3. The effect of all active transactions at the time of failure must be undone.

**Advantages:**

Immediate update allows higher concurrency because transactions write continuously to the database rather than waiting until the commit point.

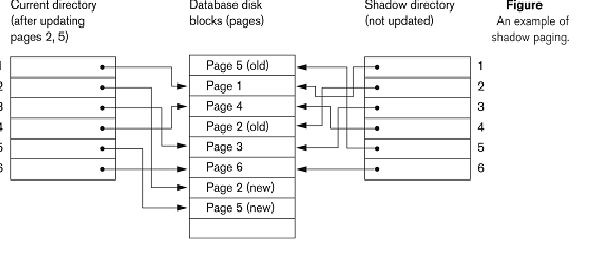
**Disadvantages:**

Step 2 above can lead to cascading rollbacks - time consuming and may be problematic.

**3. Shadow paging**

* Is a technique for providing ***automaticity*** and ***durability*** properties related to transaction control.
* This technique does not require the use of a log in a single-user environment. In a multiuser environment, a log may be needed for the concurrency control method.
* Shadow paging considers the database to be made up of a number of fixed size disk pages (or disk blocks)—say, ***n***—for recovery purposes. A **directory(page table)** with ***n***entries is constructed, where the ***i*th** entry points to the ***i*th** database page on disk.
* The idea is to maintain 2 **directories** (page tables) during the lifetime of a transaction, **Current directory** & the **Shadow** **directory**.
* The directory is kept in **main memory** if it is not too large, and all references—reads or writes—to database pages on disk go through it.
* Initially both the **directories** are identical. Only current **directory** is used for data item accesses during execution of the transaction.
* Whenever any page is about to be written for the first time-a copy of this page is made on to an unused page.
* When a transaction begins executing, the **current directory**—whose entries point to the most recent or current database pages on disk—is copied into a **shadow directory**. The shadow directory is then saved on disk while the current directory is used by the transaction.

**The figure below illustrates the use of Shadow paging techniques:**



**Advantages:**

* The overhead of maintaining the transaction log file is eliminated.
* Since there is no need for undo or redo operations, recovery is significantly fast.

**Disadvantages:**

* Copying the entire page table is very expensive.
* Complex storage management strategies.

**Authentication and Authorization**

**What is Authentication?**

The process of securely identifying its users by a system is called authentication. Authentication is used to establish the identity of a user who is trying to use a system. Establishing the identity is done by testing a unique piece of information that is known only by the user being authenticated and the authentication system. This unique piece of information could be a password, or a physical property that is unique to the user such as a fingerprint or other bio metric, etc.

The simplest and most common authentication systems used is called Local authentication methods. In this kind of a system, the usernames and password of authenticated users are stored on the local server system. When a user wants to login, he/she sends his/her username and password in plaintext to the server. It compares the received information with the database and if it is a match, the user will be authenticated.

**What is Authorization?**

An authorization scheme determines whether an authenticated user should be able to perform a particular operation on a particular resource. For example, in a database, set of users are allowed to update/ modify the database, while some users can only read the data. So, when a user logs in to the database, the authorization scheme determines whether that user should be given the ability to modify the database or just the ability to read the data. So in general, an authorization scheme determines whether an authenticated user should be able to perform a particular operation on a particular resource.