Module – 5

Chapter 1 – Transaction Processing

5.1 Introduction to Transaction Processing

- The 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 executing database transactions.
- Examples of such systems include airline reservations, banking, credit card processing, online retail purchasing, stock markets, supermarket checkouts, and many other applications.

5.1.1 Single-User versus Multiuser Systems:

- One criterion for classifying a database system is according to the number of users who can use the system concurrently.
- A DBMS is **single-user** if at most one user at a time can use the system.
- Single-user DBMSs are mostly restricted to personal computer systems; most other DBMSs are multiuser. For example, an airline reservations system is used by hundreds of users and travel agents concurrently.
- **Multiuser** if many users can use the system and access the database—concurrently.
- Database systems used in banks, insurance agencies, stock exchanges, supermarkets, and many other applications are multiuser systems.

5.1.2. Transactions, Database Items, Read and Write Operations, and DBMS Buffers

- A transaction is an executing program that forms a logical unit of database processing.
- A transaction includes one or more database access operations—these can include insertion, deletion, modification (update), or retrieval operations.
- One way of specifying the transaction boundaries is by specifying explicit **begin transaction and end transaction** statements in an application program.
- The database operations in a transaction do not update the database but only retrieve data, the transaction is called a **read-only transaction**; otherwise it is known as a **read-write transaction**

Read and write operations:

- A **database** is basically represented as a collection of named data items. The size of a data item is called its **granularity**.

- A **data item** can be a database record, but it can also be a larger unit such as a whole disk block, or even a smaller unit such as an individual field (attribute) value of some record in the database.
- The transaction processing concepts are independent of the data item granularity (size) and apply to data items in general.
- Each data item has a unique name, and it means to uniquely identify each data item.
- The basic database access operations that a transaction can include are as follows:

■ **read_item(X).** Reads a database item named X into a program variable.

■ **write_item(X).** Writes the value of program variable X into the database item named X.

Dbms Buffers:

- The basic unit of data transfer from disk to main memory is one disk page (disk block).
- Executing a **read_item(X)** command includes the following steps:

1. Find the address of the disk block that contains item X.

2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer). The size of the buffer is the same as the disk block size.

3. Copy item X from the buffer to the program variable named X

- Executing a **write_item(X)** command includes the following steps:

1. Find the address of the disk block that contains item X.

2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).

3. Copy item X from the program variable named X into its correct location in the buffer.

4. Store the updated disk block from the buffer back to disk (either immediately or at some later point in time).

- A main memory buffer is handled by the recovery manager of the DBMS in cooperation with the underlying operating system.
- The DBMS will maintain in the **database cache** a number of data buffers in main memory. Each buffer typically holds the contents of one database disk block, which contains some of the database items being processed.
- When these buffers are all occupied, and additional database disk blocks must be copied into memory, some **buffer replacement policy** is used to choose which of the current occupied buffers is to be replaced. Some commonly used buffer replacement policies are **LRU (least recently used).**

- A transaction includes read_item and write_item operations to access and update the database. Figure 20.2 shows examples of two very simple transactions.
- The read-set of a transaction is the set of all items that the transaction reads, and the write-set is the set of all items that the transaction writes.
- For example, the read-set of T1 in Figure 20.2 is {X, Y} and its write-set is also {X, Y}

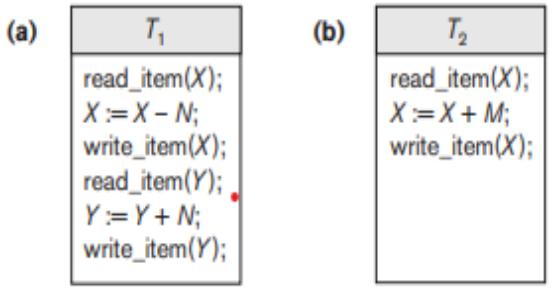


Figure 20.2
Two sample transactions.
(a) Transaction T_1.
(b) Transaction T_2.

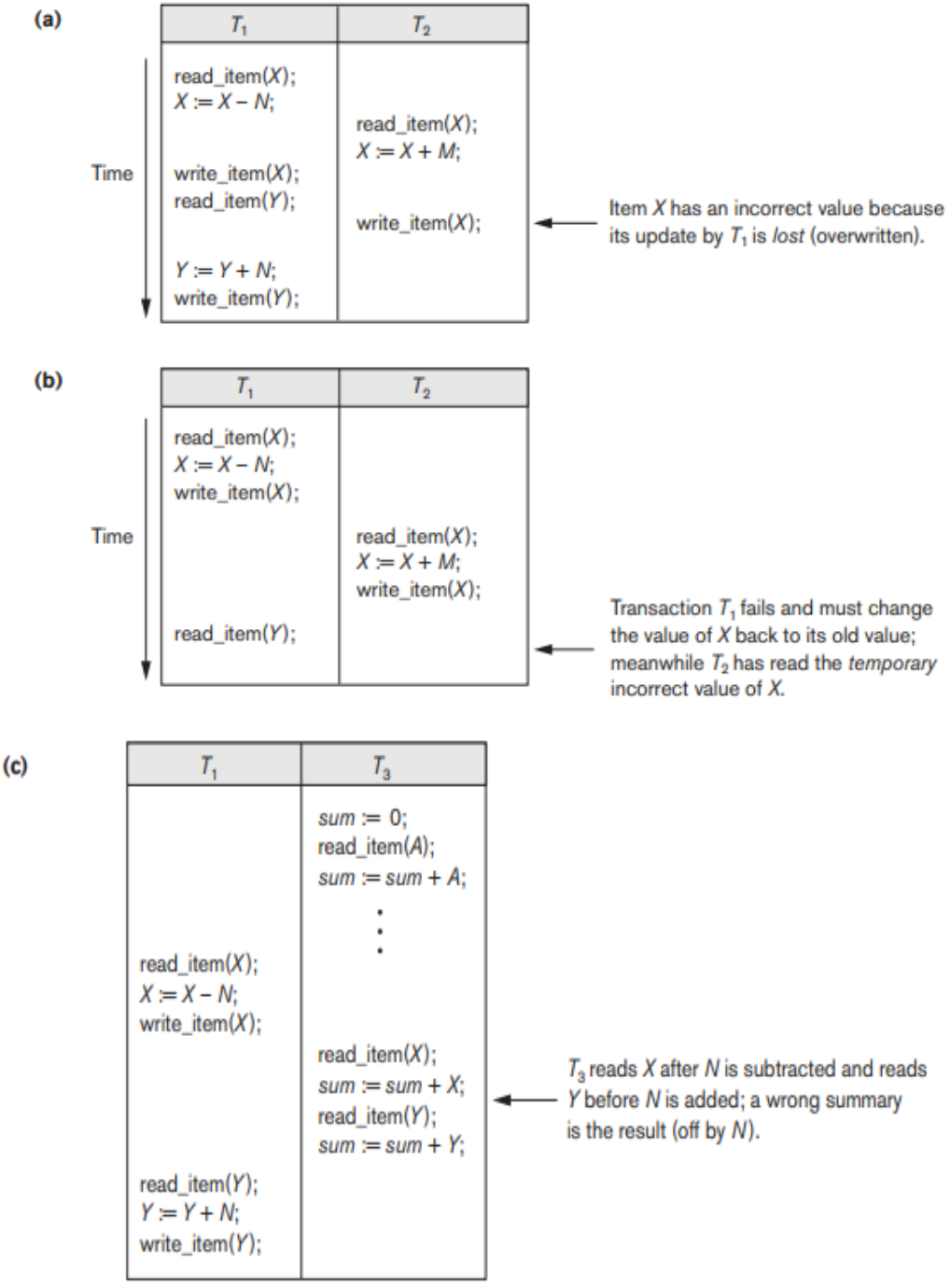
- Concurrency control and recovery mechanisms are mainly concerned with the database commands in a transaction.
- Transactions submitted by the various users may execute concurrently and may access and update the same database items.
- If this concurrent execution is uncontrolled, it may lead to problems,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

5.1.3. Why Concurrency Control Is Needed

- The types of problems we may encounter with these two simple transactions if they run concurrently:
- **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 items incorrect. Suppose that transactions T1 and T2 are submitted at approximately the same time, and suppose that their operations are interleaved.
- **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. Meanwhile, the updated item is accessed (read) by another transaction before it is changed back (or rolled back) to its original value.
- **The Incorrect Summary Problem**. If one transaction is calculating an aggregate summary function on a number of database items while other transactions are updating some of these items, the aggregate function may calculate some values before they are updated and others after they are updated.
- **The Unrepeatable Read Problem**. Another problem that may occur is called unrepeatable read, where a transaction T reads the same item twice and the item is

changed by another transaction T between the two reads. Hence, T receives different values for its two reads of the same item.

Figure 20.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.



5.1.4. Why Recovery Is Needed:

- Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either all the operations in the transaction are completed successfully and their effect is recorded permanently in the database, or that the transaction does not have any effect on the database or any other transactions.
- In the first case, the transaction is said to be **committed**, whereas in the second case, the transaction is **aborted**.
- If a transaction **fails** after executing some of its operations but before executing all of them, the operations already executed must be undone and have no lasting effect.

Types of Failures. Failures are generally classified as transaction, system, and media failures. There are several possible reasons for a transaction to fail in the middle of execution:

1. **A computer failure (system crash).** A hardware, software, or network error occurs in the computer system during transaction execution. Hardware crashes are usually media failures—for example, main memory failure.

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 erroneous parameter values or because of a logical programming error.

3**. Local errors or exception conditions detected by the transaction**. During transaction execution, certain conditions may occur that necessitate cancellation of the transaction. For example, data for the transaction may not be found. An exception condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be canceled.

4. **Concurrency control enforcement.** The concurrency control method may abort a transaction because it violates serializability or it may abort one or more transactions to resolve a state of deadlock among several transactions. Transactions aborted because of serializability violations or deadlocks are typically restarted automatically at a later time.

5. **Disk failure**. Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.

6. **Physical problems and catastrophes.** This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator

5.2 Transaction and System Concepts:

- A transaction is an atomic unit of work that should either be completed in its entirety or not done at all.

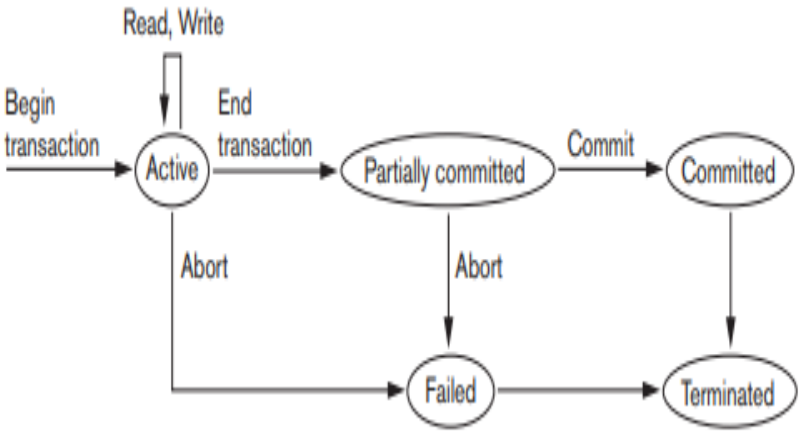
- For recovery purposes, the system needs to keep track of when each transaction starts, terminates, and commits, or aborts. Therefore, the recovery manager of the DBMS needs to keep track of the following operations:

5.2.1 Transaction States and Additional Operations:

- **begin_transaction** − A marker that specifies start of transaction execution.
- **read_item or write_item** − Database operations that may be interleaved with main memory operations as a part of transaction.
- **end_transaction** − A marker that specifies end of transaction.
- **commit** − A signal to specify that the transaction has been successfully completed in its entirety and will not be undone.
- **rollback** − A signal to specify that the transaction has been unsuccessful and so all temporary changes in the database are undone. A committed transaction cannot be rolled back.

A transaction may go through a subset of five states, active, partially committed, committed, failed and aborted.

- **Active** − The initial state where the transaction enters is the active state. The transaction remains in this state while it is executing read, write or other operations.
- **Partially Committed** − The transaction enters this state after the last statement of the transaction has been executed.
- **Committed** − The transaction enters this state after successful completion of the transaction and system checks have issued commit signal.
- **Failed** − The transaction goes from partially committed state or active state to failed state when it is discovered that normal execution can no longer proceed or system checks fail.
- **Aborted** − This is the state after the transaction has been rolled back after failure and the database has been restored to its state that was before the transaction began.



5.2.2. The System Log:

- An unique transaction-id that is generated automatically by the system for each transaction and that is used to identify each transaction:
- 1. [start_transaction, T]. Indicates that transaction T has started execution.
- 2. [write_item, T, X, old_value, new_value]. Indicates that transaction T has changed the value of database item X from old_value to new_value.
- 3. [read_item, T, X]. Indicates that transaction T has read the value of database item X.
- 4. [commit, T]. Indicates that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
- 5. [abort, T]. Indicates that transaction T has been aborted.

5.2.3 Commit Point of a Transaction:

- A transaction T reaches its **commit point** when all its operations that access the database have been executed successfully and the effect of all the transaction operations on the database have been recorded in the log.
- Beyond the commit point, the transaction is said to be committed, and its effect must be permanently recorded in the database.

5.2.4 DBMS-Specific Buffer Replacement Policies:

- The DBMS cache will hold the disk pages that contain information currently being processed in main memory buffers. If all the buffers in the DBMS cache are occupied and new disk pages are required to be loaded into main memory from disk, a page replacement policy is needed to select the particular buffers to be replaced. Some page replacement policies that have been developed specifically for database systems are:

- **Domain Separation (DS) Method.** In a DBMS, various types of disk pages exist: index pages, data file pages, log file pages, and so on. In this method, the DBMS cache is divided into separate domains (sets of buffers). Each domain handles one type of data via the basic LRU (least recently used) page replacement
- **Hot Set Method**. This page replacement algorithm is useful in queries that have to scan a set of pages repeatedly, such as when a join operation is performed using the nested-loop method. The hot set method determines for each database processing algorithm the set of disk pages that will be accessed repeatedly, and it does not replace them until their processing is completed.
- **The DBMIN Method.** This page replacement policy uses a model known as QLSM (query locality set model), which predetermines the pattern of page references for each algorithm for a particular type of database operation.

5.3 Desirable Properties of Transactions:

- Transactions should possess several properties, often called the ACID properties; they should be enforced by the concurrency control and recovery methods of the DBMS. The following are the ACID properties:

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

■ **Consistency preservation**. A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.

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

■ **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

5.4 Transaction Support in SQL:

- The basic definition of an SQL transaction is similar to our already defined concept of a transaction. That is, it is a logical unit of work and is guaranteed to be atomic.
- A single SQL statement is always considered to be atomic—either it completes execution without an error or it fails and leaves the database unchanged.
- With SQL, there is no explicit Begin_Transaction statement. Transaction initiation is done implicitly when particular SQL statements are encountered. However, every transaction must have an explicit end statement, which is either a COMMIT or a ROLLBACK.

- Every transaction has certain characteristics attributed to it. These characteristics are specified by a SET TRANSACTION statement in SQL. The characteristics are the access mode, the diagnostic area size, and the isolation level.
- The access mode can be specified as READ ONLY or READ WRITE. The default is READ WRITE, and this mode of READ WRITE allows select, update, insert, delete, and create commands to be executed. A mode of READ ONLY, as the name implies, is simply for data retrieval.
- The diagnostic area size option, DIAGNOSTIC SIZE n, specifies an integer value n, which indicates the number of conditions that can be held simultaneously in the diagnostic area. These conditions supply feedback information (errors or exceptions) to the user or program on the n most recently executed SQL statement.
- The isolation level option is specified using the statement ISOLATION LEVEL , where the value for can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, or SERIALIZABLE.
- The default isolation level is SERIALIZABLE, although some systems use READ COMMITTED as their default. The use of the term SERIALIZABLE here is based on not allowing violations that cause :

1. **dirty read**: A transaction T1 may read the update of a transaction T2, which has not yet committed. If T2 fails and is aborted, then T1 would have read a value that does not exist and is incorrect.

2. **unrepeatable read**: A transaction T1 may read a given value from a table. If another transaction T2 later updates that value and T1 reads that value again, T1 will see a different value

3. **phantoms:** A transaction T1 may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE-clause. Now suppose that a transaction T2 inserts a new row r that also satisfies the WHERE-clause condition used in T1, into the table used by T1. The record r is called a phantom record because it was not there when T1 starts but is there when T1 ends.

Table 20.1 Possible Violations Based on Isolation Levels as Defined in SQL

	Type of Violation		
Isolation Level	**Dirty Read**	**Nonrepeatable Read**	**Phantom**
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

- As we have seen, SQL provides a number of transaction-oriented features. The DBA or database programmers can take advantage of these options to try improving transaction performance by relaxing serializability if that is acceptable for their applications.
- **Snapshot Isolation:** The basic definition of snapshot isolation is that a transaction sees the data items that it reads based on the committed values of the items in the database snapshot (or database state) when the transaction starts.
- Snapshot isolation will ensure that the phantom record problem does not occur.
- Any insertions, deletions, or updates that occur after the transaction starts will not be seen by the transaction.

Chapter 2 – Concurrency Control

I. Concurrency Control technique:

- Concurrency control is a very important concept of DBMS which ensures the simultaneous execution or manipulation of data by several processes or user without resulting in data inconsistency.
- Concurrency control provides a procedure that is able to control concurrent execution of the operations in the database.
- The fundamental goal of database concurrency control is to ensure that concurrent execution of transactions does not result in a loss of database consistency. The concept of serializability can be used to achieve this goal, since all serializable schedules preserve consistency of the database.

II. Two – phase locking techniques for concurrency control:

- The formal definition of a lock is as follows:
- **A Lock is a variable assigned to any data item in order to keep track of the status of that data item so that isolation and non-interference is ensured during concurrent transactions.**
- At its basic, a database lock exists to prevent two or more database users from performing any change on the same data item at the very same time.
- Therefore, it is correct to interpret this technique as a means of synchronizing access, which is in stark contrast to other sets of protocols such as those using timestamps and multiversion timestamps.

1. **Binary Locks :**

- A binary lock is a variable capable of holding only 2 possible values, i.e., a **1 (depicting a locked state)** or a **0 (depicting an unlocked state)**.
- This lock is usually associated with every data item in the database (maybe at table level, row level or even the entire database level).
- Should item X be unlocked, then a corresponding object lock(X) would return the value 0. So, the instant a user/session begins updating the contents of item X, lock(X) is set to a value of 1.
- There are 2 operations used to implement binary locks. They are lock_data( ) and unlock_data( ). The algorithms have been discussed below:
- **lock_item(X):**

B: if LOCK(X) = 0 (*item is unlocked*)

then LOCK(X) ←1 (*lock the item*)

else

begin

wait (until LOCK(X) = 0 and the lock manager wakes up the transaction);

go to B

end;

unlock_item(X):

LOCK(X) ← 0; (* unlock the item *)

if any transactions are waiting

then wakeup one of the waiting transactions;

- **Merits of Binary Locks :**
- They are simple to implement since they are effectively mutually exclusive and establish isolation perfectly.
- Binary Locks demand less from the system since the system must only keep a record of the locked items. The system is the **lock manager subsystem** which is a feature of all DBMSs today.
- **Drawbacks of Binary Locks :**
- Binary locks are highly restrictive.
- They do not even permit reading of the contents of item X. As a result, they are not used commercially.

2. **Shared or Exclusive Locks :**

- The incentive governing these types of locks is the restrictive nature of binary locks. Here we look at locks which permit other transactions to make read queries since **a READ query is non-conflicting**. However, if a transaction demands a write query on item X, then that transaction must be given exclusive access to item X.
- We require a kind of **multi-mode lock** which is what shared/exclusive locks are. They are also known as **Read/Write locks**.
- Unlike binary locks, Read/Write locks may be set to 3 values, i.e., **SHARED**, **EXCLUSIVE** or **UNLOCKED**. Hence, our lock, i.e., lock(X), may reflect either of the following values:

- **READ-LOCKED –** If a transaction only requires to read the contents of item X and the lock only permits reading. This is also known as a ***shared lock***.
- **WRITE-LOCKED –**If a transaction needs to update or write to item X, the lock must restrict all other transactions and provide exclusive access to the current transaction. Thus, these locks are also known as ***exclusive locks***.
- **UNLOCKED –**Once a transaction has completed its read or update operations, no lock is held and the data item is unlocked. In this state, the item may be accessed by any queued transactions.
- **read_lock(X):**

B: if LOCK(X) = “unlocked”

then begin LOCK(X) ← “read-locked”;

no_of_reads(X) ← 1

end

else if LOCK(X) = “read-locked”

then no_of_reads(X) ← no_of_reads(X) + 1

else begin

wait (until LOCK(X) = “unlocked” and the lock manager wakes up the transaction);

go to B

end;

write_lock(X):

B: if LOCK(X) = “unlocked”

then LOCK(X) ← “write-locked”

else begin

wait (until LOCK(X) = “unlocked”

and the lock manager wakes up the transaction);

go to B

end

unlock (X):

if LOCK(X) = “write-locked”

then begin LOCK(X) ← “unlocked”;

wakeup one of the waiting transactions, if any

end

else it LOCK(X) = “read-locked”

then begin

no_of_reads(X) ← no_of_reads(X) −1;

if no_of_reads(X) = 0

then begin LOCK(X) = “unlocked”;

wakeup one of the waiting transactions, if any

end

end;

- Here are a few rules that Shared/Exclusive Locks must obey:

1. A transaction **T** MUST issue the unlock(X) operation after all read and write operations have finished.
2. A transaction **T** may NOT issue a read_lock(X) or write_lock(X) operation on an item which already has a read or write-lock issued to itself.
3. A transaction **T** is NOT allowed to issue the unlock(X) operation unless it has been issued with a read_lock(X) or write_lock(X) operation.

3.Two Phase Locking:

- A transaction is said to follow the Two-Phase Locking protocol if Locking and Unlocking can be done in two phases.
- **Growing Phase:** New locks on data items may be acquired but none can be released.
- **Shrinking Phase:** Existing locks may be released but no new locks can be acquired.

	T1	T2
1	lock-S(A)	
2		lock-S(A)
3	lock-X(B)	
4	.....	.....
5	Unlock(A)	
6		Lock-X(C)
7	Unlock(B)	
8		Unlock(A)
9		Unlock(C)
10	.....	.....

Problem with Two-Phase Locking

- It does not insure recoverability which can be solved by strict two-phase locking and rigorous two-phase locking.
- It does not ensure a cascade-less schedule which can be solved by strict two-phase locking and rigorous two-phase locking.
- It may suffer from deadlock which can be solved by conservative two-phase locking.

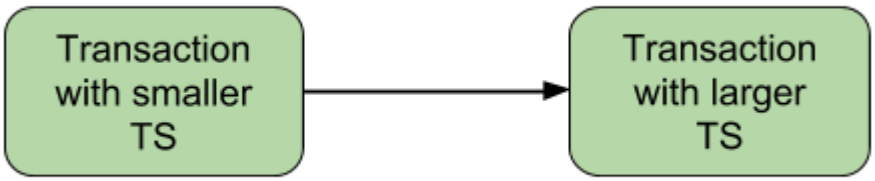
III. concurrency control based on timestamp ordering

- Timestamp-based concurrency control is a method used in database systems to ensure that transactions are executed safely and consistently without conflicts, even when multiple transactions are being processed simultaneously.
- This approach relies on timestamps to manage and coordinate the execution order of transactions. Refer to the timestamp of a transaction *T* as *TS(T)*.
- An algorithm must ensure that, for each item accessed by Conflicting Operations in the schedule, the order in which the item is accessed does not violate the ordering. To ensure this, use two Timestamp Values relating to each database item X.
- W_TS(X) is the largest timestamp of any transaction that executed write(X) successfully.
- R_TS(X) is the largest timestamp of any transaction that executed read(X) successfully.

1. Basic Timestamp Ordering:

- Every transaction is issued a timestamp based on when it enters the system. Suppose, if an old transaction T_i has timestamp TS(T_i), a new transaction T_j is assigned timestamp TS(T_j) such that **TS(T_i) < TS(T_j)**.
- The protocol manages concurrent execution such that the timestamps determine the serializability order. Whenever some Transaction *T* tries to issue a R_item(X) or a W_item(X), the Basic TO algorithm compares the timestamp of *T* with **R_TS(X) & W_TS(X)** to ensure that the Timestamp order is not violated. This describes the Basic TO protocol in the following two cases.
- Whenever a Transaction *T* issues a **W_item(X)** operation, check the following conditions:
    - If ***R_TS(X) > TS(T)*** and if ***W_TS(X) > TS(T)***, then abort and rollback T and reject the operation. else,
    - Execute W_item(X) operation of T and set W_TS(X) to TS(T).
- Whenever a Transaction *T* issues a **R_item(X)** operation, check the following conditions:
    - If ***W_TS(X) > TS(T)***, then abort and reject T and reject the operation, else
    - If W_TS(X) <= TS(T), then execute the R_item(X) operation of T and set R_TS(X) to the larger of TS(T) and current R_TS(X).
- **Disadvantages of Basic TO protocol:**

- Timestamp Ordering protocol ensures serializability since the precedence graph will be of the form:



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

2. **Strict Timestamp Ordering:**

- A variation of Basic TO is called **Strict TO** ensures that the schedules are both Strict and Conflict Serializable. In this variation, a Transaction T that issues a R_item(X) or W_item(X) such that TS(T) > W_TS(X) has its read or write operation delayed until the Transaction **T'** that wrote the values of X has committed or aborted.
- **Advantages of Timestamp Ordering Protocol**
- **High Concurrency:** Timestamp-based concurrency control allows for a high degree of concurrency by ensuring that transactions do not interfere with each other.
- **Efficient:** The technique is efficient and scalable, as it does not require locking and can handle a large number of transactions.
- **No Deadlocks:** Since there are no locks involved, there is no possibility of deadlocks occurring.
- **Improved Performance:** By allowing transactions to execute concurrently, the overall performance of the database system can be improved.
- **Disadvantages of Timestamp Ordering Protocol**
- **Limited Granularity:** The granularity of timestamp-based concurrency control is limited to the precision of the timestamp. This can lead to situations where transactions are unnecessarily blocked, even if they do not conflict with each other.
- **Timestamp Ordering:** In order to ensure that transactions are executed in the correct order, the timestamps need to be carefully managed. If not managed properly, it can lead to inconsistencies in the database.
- **Timestamp Synchronization:** Timestamp-based concurrency control requires that all transactions have synchronized clocks. If the clocks are not synchronized, it can lead to incorrect ordering of transactions.

- **Timestamp Allocation:** Allocating unique timestamps for each transaction can be challenging, especially in distributed systems where transactions may be initiated at different locations.

3. Thomas's write rule

- A modification of the basic TO algorithm, known as Thomas's write rule, does not enforce conflict serializability, but it rejects fewer write operations by modifying the checks for the write_item(X) operation as follows:
    - If read_TS(X) > TS(T), then abort and roll back T and reject the operation.
    - If write_TS(X) > TS(T), then do not execute the write operation but continue processing.
- This is because some transaction with timestamp greater than TS(T)—and hence after T in the timestamp ordering—has already written the value of X. Thus, we must ignore the write_item(X) operation of T because it is already outdated and obsolete. Notice that any conflict arising from this situation would be detected by case (1).
    - If neither the condition in part (1) nor the condition in part (2) occurs, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).

IV. Multiversion Concurrency Control Techniques

- Multi-Version Concurrency Control (MVCC) is a database optimization method, that makes redundant copies of records to allow for safe concurrent reading and updating of data. DBMS reads and writes are not blocked by one another while using MVCC. A technique called concurrency control keeps concurrent processes running to avoid read/write conflicts or other irregularities in a database.
- Multi-Version Concurrency Control is a technology, utilized to enhance databases by resolving concurrency problems and also data locking by preserving older database versions. When many tasks attempt to update the same piece of data simultaneously, MVCC causes a conflict and necessitates a retry from one or more of the processes.

Advantages of Multi-Version Concurrency Control (MVCC) in DBMS:

- Below are some advantages of Multi-Version Concurrency Control in DBMS
- **The reduced read-and-write necessity for database locks:** The database can support many read-and-write transactions without locking the entire system thanks to MVCC.
- **Increased Concurrency:** This Enables several users to use the system at once.
- **Minimize read operation delays:** By enabling concurrent read operations, MVCC helps to cut down on read times for data.

- **Accuracy and consistency:** Preserves data integrity over several concurrent transactions.

Disadvantages of Multi-Version Concurrency Control (MVCC) in DBMS:

- Below are some disadvantages of Multi-Version Concurrency Control in DBMS
- **Overhead:** Keeping track of many data versions might result in overhead.
- **Garbage collecting:** To get rid of outdated data versions, MVCC needs effective garbage collecting systems.
- **Increase the size of the database:** Expand the capacity of the database since MVCC generates numerous copies of the records and/or tuples.
- **Complexity:** Compared to more straightforward locking systems, MVCC usage might be more difficult.

1. **Multiversion technique based on Timestamp Ordering:**

- In the Multiversion timestamp ordering technique, for each transaction in the system, a unique timestamp is assigned before the start of the execution of the transaction. The timestamp of a transaction T is denoted by TS(T). For each data item X a sequence of versions $<X_1, X_2, X_3, \ldots\ldots X_K>$ is associated.
- For each version X_i of a data item(X), the system maintains the following three fields:

1. The **value** of the version.

2. **Read_TS (X_i):** The read timestamp of X_i is the largest timestamp of any transaction that successfully reads version X_i.

3. **Write_TS(X_i):** The write timestamp of X_i is the largest timestamp of any transaction that successfully writes version X_i.

- **To ensure serializability the following two rules are used:**
- Suppose a transaction T issues a read request and a written request for a data item X. Let X_i be the version having the largest Write_TS(X_i) of all the versions of X which is also less than or equal to TS(T).
- **Rule1:** Let us suppose the transaction T issues a Read(X) request, If Read_TS(X_i)<TS(T) then the system returns the value of X_i to the transaction T and update the value of Read_TS(X_i) to TS(T)
- **Rule 2:** Let us suppose the transaction T issues a Write(X) request TS(T) < Read_TS(X), then the system aborts transaction T. On the other hand, if TS(T) = Write_TS(X), the system overwrites the contents of X; if TS(T)>Write_TS(X) it creates a new version of X.

2.Multiversion Two-Phase Locking Using Certify Locks:

- In this multiple-mode locking scheme, there are *three locking modes* for an item: read, write, and *certify*.
- Hence, the state of LOCK(*X*) for an item *X* can be one of read-locked, write-locked, certify-locked, or unlocked.
- We can describe the relationship between read and write locks in the standard scheme by means of the lock compatibility table shown in Figure 22.6(a).
- An entry of *Yes* means that if a transaction *T* holds the type of lock specified in the column header on item *X* and if transaction *T* requests the type of lock specified in the row header on the same item *X*, then *T can obtain the lock* because the locking modes are compatible.
- On the other hand, an entry of *No* in the table indicates that the locks are not compatible, so *T must wait* until *T releases* the lock.

(a)

	Read	Write
Read	Yes	No
Write	No	No

(b)

	Read	Write	Certify
Read	Yes	Yes	No
Write	Yes	No	No
Certify	No	No	No

Figure 22.6
Lock compatibility tables. (a) A compatibility table for read/write locking scheme. (b) A compatibility table for read/write/certify locking scheme.

- In this multiversion 2PL scheme, reads can proceed concurrently with a single write operation—an arrangement not permitted under the standard 2PL schemes.
- The cost is that a transaction may have to delay its commit until it obtains exclusive certify locks on *all the items* it has updated.
- It can be shown that this scheme avoids cascading aborts, since transactions are only allowed to read the version *X* that was written by a committed transaction.
- However, deadlocks may occur if upgrading of a read lock to a write lock is allowed, and these must be handled by variations of the techniques described.

V. Validation (Optimistic) Concurrency Control technique:

- This protocol is used in DBMS (Database Management System) for avoiding concurrency in transactions. It is called optimistic because of the assumption it makes, i.e. very less interference occurs, therefore, there is no need for checking while the transaction is executed.
- In this technique, no checking is done while the transaction is been executed. Until the transaction end is reached updates in the transaction are not applied directly to the database.
- All updates are applied to local copies of data items kept for the transaction.
- At the end of transaction execution, while execution of the transaction, a validation phase checks whether any of transaction updates violate serializability. If there is no violation of serializability the transaction is committed and the database is updated; or else, the transaction is updated and then restarted.
- Optimistic Concurrency Control is a three-phase protocol. The three phases for validation based protocol:
- **Read Phase:**
  Values of committed data items from the database can be read by a transaction. Updates are only applied to local data versions.
- **Validation Phase:**
  Checking is performed to make sure that there is no violation of serializability when the transaction updates are applied to the database.
- **Write Phase:**
  On the success of the validation phase, the transaction updates are applied to the database, otherwise, the updates are discarded and the transaction is slowed down.
- In order to perform the Validation test, each transaction should go through the various phases as described above. Then, we must know about the following three time-stamps that we assigned to transaction T_i, to check its validity:
- **1. Start(T_i):** It is the time when T_i started its execution.
- **2. Validation(T_i):** It is the time when T_i just finished its read phase and begin its validation phase.
- **3. Finish(T_i):** the time when T_i end it's all writing operations in the database under write-phase.
- Two more terms that we need to know are:

1. Write_set: of a transaction contains all the write operations that T_i performs.

2. Read_set: of a transaction contains all the read operations that T_i performs.

- In the Validation phase for transaction T_i the protocol inspect that T_i doesn't overlap or intervene with any other transactions currently in their validation phase or in committed. The validation phase for T_i checks that for all transaction T_j one of the following below conditions must hold to being validated or pass validation phase:
- **1. Finish(T_j)<Starts(T_i)**, since T_j finishes its execution means completes its write-phase before T_i started its execution(read-phase). Then the serializability indeed maintained.
- **2.** T_i begins its write phase after T_j completes its write phase, and the read_set of T_i should be disjoint with write_set of T_j.
- **3.** T_j completes its read phase before T_i completes its read phase and both read_set and write_set of T_i are disjoint with the write_set of T_j.