Module - III

Transaction Management and Concurrency Control

Q1. What is a transaction.

In database terms, a **transaction** is any action that reads from and/or writes to a database. A transaction may consist of a simple SELECT statement to generate a list of table contents; it may consist of a series of related UPDATE statements to change the values of attributes in various tables; it may consist of a series of INSERT statements to add rows to one or more tables, or it may consist of a combination of SELECT, UPDATE, and INSERT statements.

More formally, **transaction** is a logical unit of work that must be entirely completed or entirely aborted; no intermediate states are acceptable. That is, all of the SQL statements in the transaction must be completed successfully. If any of the SQL statements fail, the entire transaction is rolled back to the original database state that existed before the transaction started. A successful transaction changes the database from one consistent state to another. A **consistent database state** is one in which all data integrity constraints are satisfied.

Q2. Explain the desirable properties of transactions.

Transactions should possess several properties, often called the ACID properties. In addition, when executing multiple transactions, the DBMS must schedule the concurrent execution of the transaction's operations. The schedule of such transaction's operations must exhibit the property of serializability. The following are the properties:

1. **Atomicity** requires that all operations (SQL requests) of a transaction be completed; if not, the transaction is aborted. In other words, a transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.

2. **Consistency** indicates the permanence of the database's consistent state. A transaction takes a database from one consistent state to another consistent state. When a transaction is completed, the database must be in a consistent state; if any of the transaction parts violates an integrity constraint, the entire transaction is aborted.

3. **Isolation** means that the data used during the execution of a transaction cannot be used by a second transaction until the first one is completed. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently. This property is particularly useful in multiuser database environments because several users can access and update the database at the same time.

4. **Durability** ensures that once transaction changes are done (committed), they cannot be undone or lost, even in the event of a system failure.

5. Serializability ensures that the schedule for the concurrent execution of the transactions yields consistent results. This property is important in multiuser and distributed databases, where multiple transactions are likely to be executed concurrently.

Q3. Explain the different states for the execution of transaction by using a suitable diagram.

A transaction is atomic unit of work that is either completed in its entirety or not done at all. A state transition diagram describes how a transaction moves through its execution states. A transaction

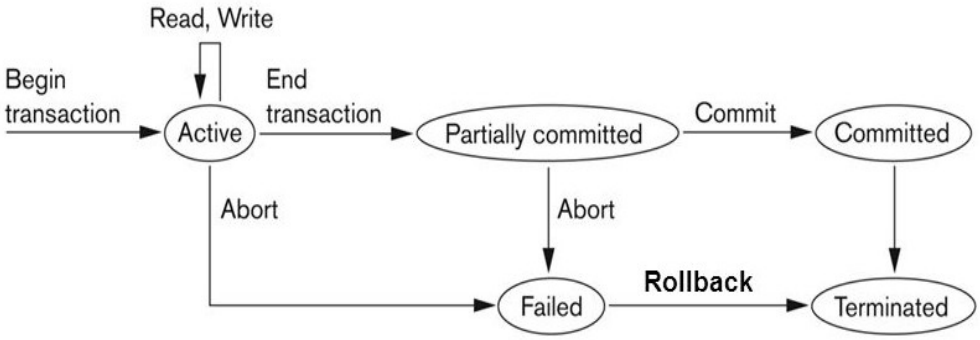

Figure 12: State transition diagram illustrating the states for transaction execution

goes into an **active state** immediately after it starts execution, where it can issue READ and WRITE operations. When the transaction ends, it moves to the **partially committed state**. At this point, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently. Once this check is successful, the transaction is said to have reached its commit point and enters the **committed state**. Once a transaction is committed, it has concluded its execution successfully and all its changes must be recorded permanently in the database. However, a transaction can go to the **failed state** if one of the checks fails or if the transaction is aborted during its active sate. The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database. The **terminated state** corresponds to the transaction leaving the system. Failed or aborted transactions may be restarted later-either automatically or after being resubmitted by the user-as brand new transactions.

Q4. Explain the transaction management in DBMS using SQL.

A transaction begins implicitly when the first SQL statement is encountered or a transaction management statements such as BEGIN TRANSACTION is encountered. Once a transaction sequence is initiated, the sequence must continue through all succeeding SQL statements until one of the following four events occurs:

1. A COMMIT statement is reached, in which case all changes are permanently recorded within the database. The COMMIT statement automatically ends the SQL transaction.

2. A ROLLBACK statement is reached, in which case all changes are aborted and the database is rolled back to its previous consistent state.

3. The end of a program is successfully reached, in which case all changes are permanently recorded within the database. This action is equivalent to COMMIT.

4. The program is abnormally terminated, in which case the changes made in the database are aborted and the database is rolled back to its previous consistent state. This action is equivalent to ROLLBACK.

Q5. Explain transaction LOG and its use in managing the transaction.

A DBMS uses a **transaction log** to keep track of all transactions that update the database. The information stored in this log is used by the DBMS for a recovery requirement triggered by a ROLLBACK statement, a program's abnormal termination, or a system failure such as a network discrepancy or a disk crash.

While the DBMS executes transactions that modify the database, it also automatically updates the transaction log. The transaction log stores:

1. A record for the beginning of the transaction.
2. For each transaction component (SQL statement):
   (a) The type of operation being performed (update, delete, insert).
   (b) The names of the objects affected by the transaction (the name of the table).
   (c) The "before" and "after" values for the fields being updated.
   (d) Pointers to the previous and next transaction log entries for the same transaction.
3. The ending (COMMIT) of the transaction.

An example log file is given in Figure 13:

TRL ID	TRX NUM	PREV PTR	NEXT PTR	OPERATION	TABLE	ROW ID	ATTRIBUTE	BEFORE VALUE	AFTER VALUE
341	101	Null	352	START	****Start Transaction				
352	101	341	363	UPDATE	CUSTOMER	1558-QW1	CUST_BALANCE	2001.00	1001.00
363	101	352	365	UPDATE	CUSTOMER	10011	CUST_BALANCE	1525.00	2525.00
365	101	363	Null	COMMIT	**** End of Transaction				

TRL_ID = Transaction log record ID
TRX_NUM = Transaction number
PTR = Pointer to a transaction log record ID
(Note: The transaction number is automatically assigned by the DBMS.)

Figure 13: A Transaction Log

The transaction log is a critical part of the database, and it is usually implemented as one or more files that are managed separately from the actual database files.

Concurrency Control Techniques

Q6. Explain why concurrency control is need in DBMS.

The coordination of the simultaneous execution of transactions in a multiuser database system is known as **concurrency control**. The objective of concurrency control is to ensure the serializability of transactions in a multiuser database environment. Concurrency control is important because

the simultaneous execution of transactions over a shared database can create several data integrity and consistency problems. If this concurrent execution is uncontrolled, it may lead to the following problems:

T_1	T_2
read_item(X);	
X := X - N;	
	read_item(X);
	X := X + M;
write_item(X);	
read_item(Y);	
	write_item(X);
Y := Y + N;	
write_item(Y);	

The Lost Updates 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 T_1 and T_2 are submitted at approximately the same time, and suppose that their operations are interleaved; then the final value of item X is incorrect because T_2 reads the value of X before T_1 changes it in the database, and hence the updated value resulting from T_1 is lost.

Uncommitted Data This problem occurs when one transactions 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. Suppose that transactions T_1 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 T_2 reads the temporary value of X, which will not be recorded permanently in the database because of the failure of T_1. Therefore, the value of item X that is read by T_2 is dirty because it has been created by a transaction that has not completed and committed yet.

T_1	T_2
read_item(X);	
X:= X - N;	
read_item(X);	
	read_item(X);
	X := X + M;
	write_item(X);
read_item(Y);	

T_1	T_2
	sum := 0;
	read_item(A);
	sum := sum + A;
	⋮
read_item(X);	
X := X - N;	
write_item(X);	
	read_item(Y);
	sum := sum + X;
	read_item(Y);
	sum := sum + Y;
read_item(Y);	
Y := Y + N;	
write_item(Y);	

Inconsistent Retrievals 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. Suppose that a transactions T_2 is calculating the total number of reservations on all the flights; meanwhile the transaction T_1 is executing. Since the operations are interleaved, the result of T_2 will be off by an amount N because T_2 reads the value of X after N seats have been subtracted from it but reads the value of Y before those N seats have been added to it.

Q7. Explain serial, non-serial and serializable schedule.

Table 1: Schedule A

T_1	T_2
read_item(X);	
X := X - N;	
write_item(X);	
read_item(Y);	
Y := Y + N;	
write_item(Y);	
	read_item(X);
	X := X + M;
	write_item(X);

Serial Schedule: Schedule A is called serial because the operations of each transaction are executed consecutively, without any interleaved operations from the other transaction. In a serial schedule, entire transactions are performed in serial order: T_1 then T_2 as in the schedule A. Therefore, in a serial schedule, only one transaction at a time is active - the commit or abort of the active transaction initiates execution of the next transaction. Since all the transactions are independent in the serial schedule, every serial schedule is considered to be correct.

Non-Serial Schedule: A schedule S is non-serial if, for every transaction T participating in the schedule, all the operations of T are interleaved. Therefore, all the transactions in the schedule are active in a non-serial schedule. Advantage of non-serial schedule is that one transaction must not wait too long to execute. However, since all the transactions are interleaved, results of the non-serial schedules are unpredictable as in the schedule B.

Table 2: Schedule B

T_1	T_2
read_item(X);	
X:= X - N;	
	read_item(X);
	X := X + M;
write_item(X);	
read_item(Y);	
	write_item(X);
Y := Y + N;	
write_item(Y);	

Table 3: Schedule C

T_1	T_2
read_item(X);	
X := X - N;	
write_item(X);	
	read_item(X);
	X := X + M;
	write_item(X);
read_item(Y);	
Y := Y + N;	
write_item(Y);	

Serializable Schedule: A schedule S is said to be **serializable schedule** when all the transactions in the schedule performs in non-serial and yet whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over S. 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. Therefore, a schedule S of n transactions is **serializable** if it is equivalent to some serial schedule of the same n transactions. There are two types of Serializability: Conflict Serializability and View Serializability.

Q8. Explain what do you mean by a scheduler and its objective in the transaction management.

The **scheduler** is a special DBMS process that establishes the order in which the operations within concurrent transactions are executed. The scheduler interleaves the execution of database operations to ensure serializability and isolation of transactions. However, not all transactions are serializable. The scheduler's main job is to create a serializable schedule of a transaction's operations. The scheduler also makes sure that the computer's central processing unit (CPU) and

storage systems are used efficiently. Additionally, the scheduler facilitates data isolation to ensure that two transactions do not update the same data element at the same time.

Q9. Explain locks and lock granularity in DBMS.

A **lock** guarantees exclusive use of a data item to a current transaction. That is, a transaction T_2 does not have access to a data item that is currently being used by transaction T_1. A transaction acquires a lock prior to data access; the lock is released (unlocked) when the transaction is completed so that another transaction can lock the data item for its exclusive use.

Lock granularity indicates the level of lock use. Locking can take place at the following levels: database, table, page, row, or even field (attribute).

Database Level

In a **database-level lock**, the entire database is locked, thus preventing the use of any tables in the database by transaction T_2 while transaction T_1 is being executed. This level of locking is good for batch processes, but it is unsuitable for multiuser DBMSs. Figure 14 illustrates the database-level lock. In this example, because of the database-level lock, transactions T_1 and T_2 cannot access the same database concurrently even when they use different tables.

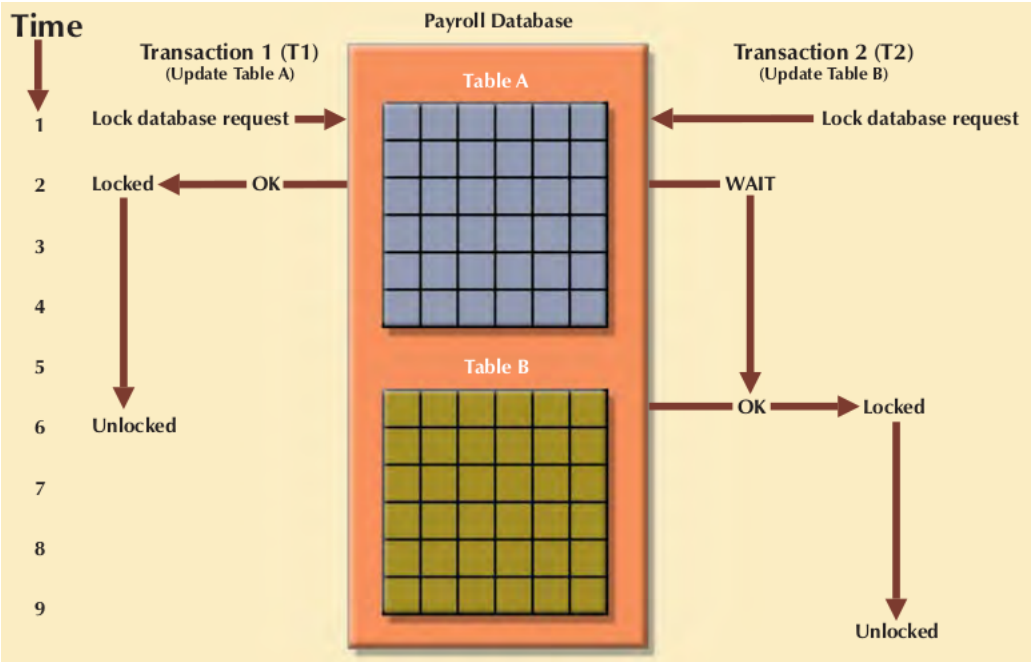

Figure 14: Database-level locking sequence

Table Level

In a **table-level lock**, the entire table is locked, preventing access to any row by transaction T_2 while transaction T_1 is using the table (See Figure 15). If a transaction requires access to several tables, each table may be locked. However, two transactions can access the same database as long as they access different tables. Table-level locks, while less restrictive than database-level locks, cause traffic jams when many transactions are waiting to access the same table.

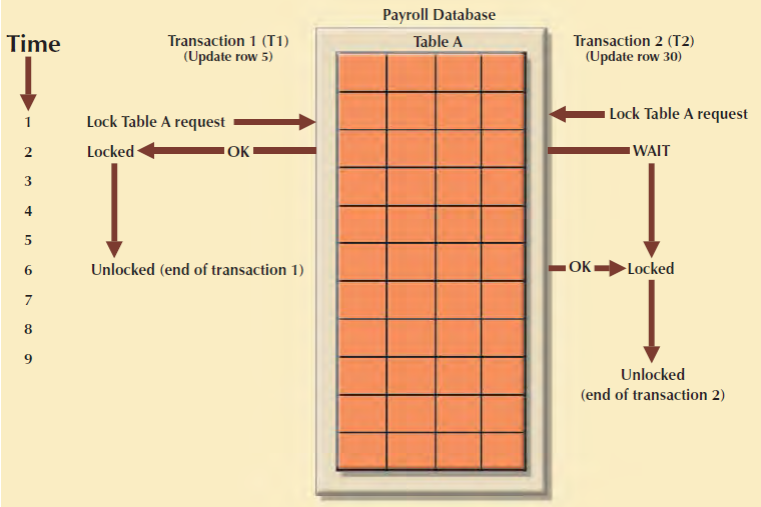

Figure 15: An example of a table-level lock

Page Level

In a **page-level lock**, the DBMS will lock an entire diskpage. A diskpage, or page, is the equivalent of a diskblock, which can be described as a directly addressable section of a disk. A table can span several pages, and a page can contain several rows of one or more tables. Page-level locks are currently the most frequently used multiuser DBMS locking method. An example of a page-level lock is shown in Figure 16. Note that T_1 and T_2 access the same table while locking different diskpages. If T_2 requires the use of a row located on a page that is locked by T_1, T_2 must wait until the page is unlocked by T_1.

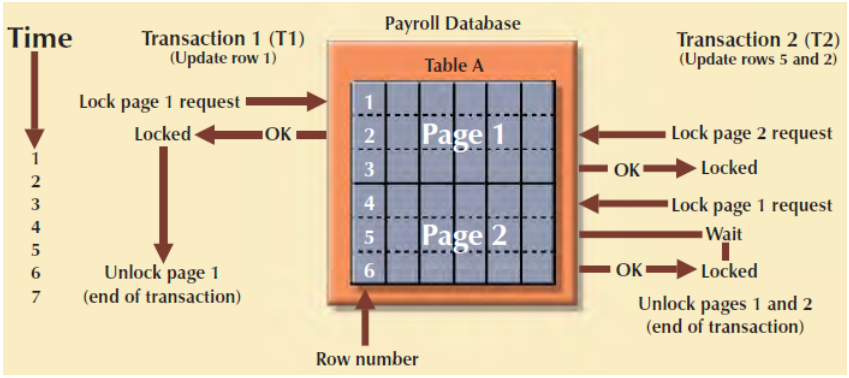

Figure 16: An example of a page-level lock

Row Level

A **row-level lock** is much less restrictive than the locks discussed earlier. The DBMS allows concurrent transactions to access different rows of the same table even when the rows are located on the same page. Although the row-level locking approach improves the availability of data, its management requires high overhead because a lock exists for each row in a table of the database involved in a conflicting transaction. Figure 17 illustrates the use of a row-level lock. Here, both

transactions can execute concurrently, even when the requested rows are on the same page. T_2 must wait only if it requests the same row as T_1.

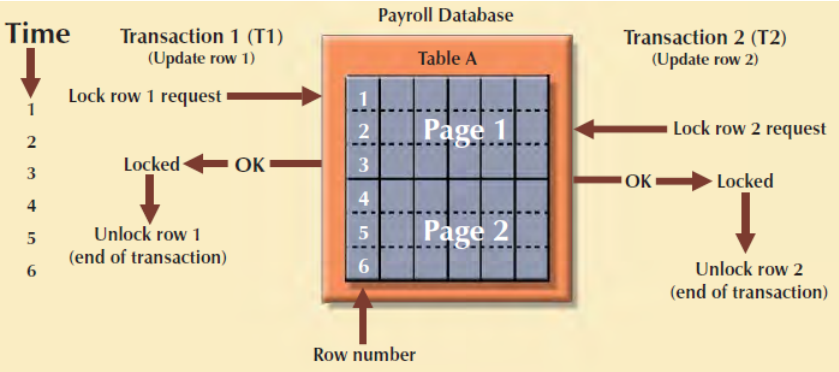

Figure 17: An example of a row-level lock

Field Level

The **field-level lock** allows concurrent transactions to access the same row as long as they require the use of different fields (attributes) within that row. Although field-level locking clearly yields the most flexible multiuser data access, it is rarely implemented in a DBMS because it requires an extremely high level of computer overhead.

Q10. Explain different types of locks used for concurrency control.

Binary Locks

A **binary lock** has only two states: locked (1) or unlocked (0). If an object—that is, a database, table, page, or row—is locked by a transaction, no other transaction can use that object. If an object is unlocked, any transaction can lock the object for its use. Every database operation requires that the affected object be locked. As a rule, a transaction must unlock the object after its termination. Therefore, every transaction requires a *lock* and *unlock* operation for each data item that is accessed. For example, in the Figure 18, DBMS will not allow two transactions to read the same database object even though neither transaction updates the database, and therefore, no concurrency problems can occur. However, binary locks are now considered too restrictive to yield optimal concurrency conditions.

TIME	TRANSACTION	STEP	STORED VALUE
1	T1	Lock PRODUCT	
2	T1	Read PROD_QOH	15
3	T1	PROD_QOH = 15 + 10	
4	T1	Write PROD_QOH	25
5	T1	Unlock PRODUCT	
6	T2	Lock PRODUCT	
7	T2	Read PROD_QOH	23
8	T2	PROD_QOH = 23 − 10	
9	T2	Write PROD_QOH	13
10	T2	Unlock PRODUCT	

Figure 18: An Example of a Binary Lock

Shared/Exclusive Locks

The labels "shared" and "exclusive" indicate the nature of the lock. An **exclusive lock** exists when access is reserved specifically for the transaction that locked the object. The exclusive lock must be used when the potential for conflict exists. A **shared lock** exists when concurrent transactions are granted read access on the basis of a common lock. A shared lock produces no conflict as long as all the concurrent transactions are read-only.

A shared lock is issued when a transaction wants to read data from the database and no exclusive lock is held on that data item. An exclusive lock is issued when a transaction wants to update (write) a data item and no locks are currently held on that data item by any other transaction. Using the shared/exclusive locking concept, a lock can have three states: *unlocked*, *shared (read)*, and *exclusive (write)*.

Note: If transaction T_2 updates data item X, an exclusive lock is required by T_2 over data item X. The exclusive lock is granted if and only if no other locks are held on the data item. Therefore, if a shared or exclusive lock is already held on data item X by transaction T_1, an exclusive lock cannot be granted to transaction T_2 and T_2 must wait to begin until T_1 commits. This condition is known as the **mutual exclusive rule**: only one transaction at a time can own an exclusive lock on the same object.

Although the use of shared locks renders data access more efficient, a shared/exclusive lock schema increases the lock manager's overhead, for several reasons:

1. The type of lock held must be known before a lock can be granted.
2. Three lock operations exist: READ_LOCK (to check the type of lock), WRITE_LOCK (to issue the lock), and UNLOCK (to release the lock).
3. The schema has been enhanced to allow a lock upgrade (from shared to exclusive) and a lock downgrade (from exclusive to shared).

Q11. Explain two phase locking protocol and how it guarantees serializability.

Two-phase locking defines how transactions acquire and relinquish locks. Two-phase locking guarantees serializability, but it does not prevent deadlocks. The two phases are:

1. A *growing phase*, in which a transaction acquires all required locks without unlocking any data. Once all locks have been acquired, the transaction is in its locked point.
2. A *shrinking phase*, in which a transaction releases all locks and cannot obtain any new lock.

The two-phase locking protocol is governed by the following rules:

- Two transactions cannot have conflicting locks.
- No unlock operation can precede a lock operation in the same transaction.
- No data are affected until all locks are obtained—that is, until the transaction is in its locked point.

In the Figure 19, the transaction acquires all of the locks it needs until it reaches its locked point. When the locked point is reached, the data are modified to conform to the transaction's requirements. Finally, the transaction is completed as it releases all of the locks it acquired in the first phase.

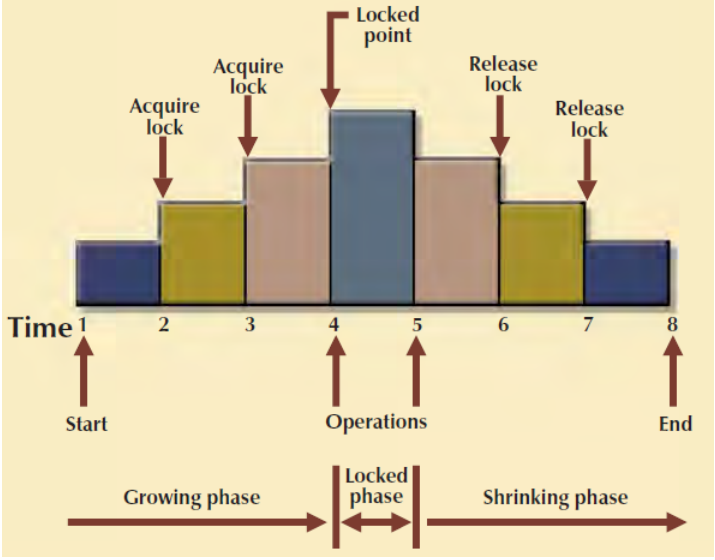

Figure 19: Two-phase locking protocol

Two-phase locking protocol can guarantee serializability. Consider the transactions T_1 and T_2 in Table 4. Transactions follow the two-phase locking protocol as all the locks precedes the unlock statements in both the transactions. It is proved that, if every transaction in a schedule follow the two-phase locking protocol, the schedule is guaranteed to be serializable, without the need to test for serializability of schedules.

Table 4: Two serializable transactions

T_1	T_2
read_lock(Y);	
read_item(Y);	
write_lock(X);	
	read_lock(X);
	read_item(X);
unlock(Y);	
read_item(X);	
X := X + Y;	
	write_lock(Y);
	unlock(X);
	read_item(Y);
	Y := X + Y;
write_item(X);	
unlock(X);	
	write_item(Y);
	unlock(Y);

Table 5: 2PL that produce deadlock

T_1	T_2
read_lock(Y);	
read_item(Y);	
	read_lock(X);
	read_item(X);
write_lock(X);	
unlock(Y);	
read_item(X);	
X := X + Y;	
	write_lock(Y);
	unlock(X);
	read_item(Y);
	Y := X + Y;
write_item(X);	
unlock(X);	
	write_item(Y);
	unlock(Y);

However, there are two limitations while using the two-phase locking protocol. These are: Two phase locking may limit the amount of concurrency that can occur in a schedule. Two-phase locking protocol may produce deadlock. Deadlock occurs when each transaction T in a set of two or

more transactions is waiting for some item that is locked by some other transaction T' in the set. This scenario is shown in the Table 5 where T_1 locked an item Y and require the other item X to proceed. Meanwhile, the transaction T_2 locked the item X and it requires the other item Y to proceed. Indeed, the two transactions T_1 and T_2 are in deadlock, as it require items that are locked by other transactions to proceed and that will never happen.

Q12. What do you mean by a deadlock. Explain three basic techniques to control deadlocks.

A deadlock occurs when two transactions wait indefinitely for each other to unlock data. For example, a deadlock occurs when two transactions, T_1 and T_2, exist in the following mode:
T_1 = access data items X and Y
T_2 = access data items Y and X
If T_1 has not unlocked data item Y, T_2 cannot begin; if T_2 has not unlocked data item X, T_1 cannot continue. Consequently, T_1 and T_2 each wait for the other to unlock the required data item.

The three basic techniques to control deadlocks are:

- *Deadlock prevention.* A transaction requesting a new lock is aborted when there is the possibility that a deadlock can occur. If the transaction is aborted, all changes made by this transaction are rolled back and all locks obtained by the transaction are released. The transaction is then rescheduled for execution. Deadlock prevention works because it avoids the conditions that lead to deadlocking.

- *Deadlock detection.* The DBMS periodically tests the database for deadlocks. If a deadlock is found, one of the transactions (the "victim") is aborted (rolled back and restarted) and the other transaction continues.

- *Deadlock avoidance.* The transaction must obtain all of the locks it needs before it can be executed. This technique avoids the rolling back of conflicting transactions by requiring that locks be obtained in succession. However, the serial lock assignment required in deadlock avoidance increases action response times.

Q13. Explain how the database concurrency is controlled with time stamping methods.

The **time stamping** approach to scheduling concurrent transactions assigns a global, unique time stamp to each transaction. The time stamp value produces an explicit order in which transactions are submitted to the DBMS. Time stamps must have two properties: uniqueness and monotonicity. **Uniqueness** ensures that no equal time stamp values can exist, and **monotonicity** ensures that time stamp values always increase.

All database operations (Read and Write) within the same transaction must have the same time stamp. The DBMS executes conflicting operations in time stamp order, thereby ensuring serializability of the transactions. If two transactions conflict, one is stopped, rolled back, rescheduled, and assigned a new time stamp value.

The disadvantage of the time stamping approach is that each value stored in the database requires two additional time stamp fields: one for the last time the field was read and one for the last update. Time stamping demands a lot of system resources because many transactions might have to be stopped, rescheduled, and restamped.

There are two schemes used to decide which transaction is rolled back and which continues executing: the wait/die scheme and the wound/wait scheme. For example, Assume that there are

two conflicting transactions: T_1 and T_2, each with a unique time stamp. Suppose T_1 has a time stamp of 11548789 and T_2 has a time stamp of 19562545. You can deduce from the time stamps that T_1 is the older transaction (the lower time stamp value) and T_2 is the newer transaction. Given that scenario, the four possible outcomes are shown in Figure 20.

TRANSACTION REQUESTING LOCK	TRANSACTION OWNING LOCK	WAIT/DIE SCHEME	WOUND/WAIT SCHEME
T1 (11548789)	T2 (19562545)	• T1 waits until T2 is completed and T2 releases its locks.	• T1 preempts (rolls back) T2. • T2 is rescheduled using the same time stamp.
T2 (19562545)	T1 (11548789)	• T2 dies (rolls back). • T2 is rescheduled using the same time stamp.	• T2 waits until T1 is completed and T1 releases its locks.

Figure 20: Wait/Die and Wound/Wait Concurrency Control Schemes

Using the wait/die scheme:

- If the transaction requesting the lock is the older of the two transactions, it will wait until the other transaction is completed and the locks are released.
- If the transaction requesting the lock is the younger of the two transactions, it will die (roll back) and is rescheduled using the same time stamp.

In short, in the wait/die scheme, the older transaction waits for the younger to complete and release its locks.

In the wound/wait scheme:

- If the transaction requesting the lock is the older of the two transactions, it will preempt (wound) the younger transaction (by rolling it back). T_1 preempts T_2 when T_1 rolls back T_2. The younger, preempted transaction is rescheduled using the same time stamp.
- If the transaction requesting the lock is the younger of the two transactions, it will wait until the other transaction is completed and the locks are released.

In short, in the wound/wait scheme, the older transaction rolls back the younger transaction and reschedules it.

Q14. Explain how the database concurrency is controlled with optimistic methods.

The **optimistic approach** is based on the assumption that the majority of the database operations do not conflict. The optimistic approach requires neither locking nor time stamping techniques. Instead, a transaction is executed without restrictions until it is committed. Using an optimistic approach, each transaction moves through two or three phases, referred to as *read*, *validation*, and *write*.

- During the *read phase*, the transaction reads the database, executes the needed computations, and makes the updates to a private copy of the database values. All update operations of the transaction are recorded in a temporary update file, which is not accessed by the remaining transactions.

- During the *validation phase*, the transaction is validated to ensure that the changes made will not affect the integrity and consistency of the database. If the validation test is positive, the transaction goes to the write phase. If the validation test is negative, the transaction is restarted and the changes are discarded.
- During the *write phase*, the changes are permanently applied to the database.

Recovery Management

Q15. Explain why recovery is needed in DBMS.

Database recovery restores a database from a given state (usually inconsistent) to a previously consistent state. 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 transaction does not affect the database or any other transactions. Contradiction to this may happen if a transaction fails after executing some of its operations but before executing all of them. There are several possible reasons for a transaction to fail in the middle of execution:

1. **Hardware/software failures.** A failure of this type could be a hard disk media failure, a bad capacitor on a motherboard, or a failing memory bank. Other causes of errors under this category include application program or operating system errors that cause data to be overwritten, deleted, or lost.
2. **Human-caused incidents.** This type of event can be categorized as unintentional or intentional.
    - An unintentional failure is caused by carelessness by end users. Such errors include deleting the wrong rows from a table, pressing the wrong key on the keyboard, or shutting down the main database server by accident.
    - Intentional events are of a more severe nature and normally indicate that the company data are at serious risk. Under this category are security threats caused by hackers trying to gain unauthorized access to data resources and virus attacks caused by disgruntled employees trying to compromise the database operation and damage the company.
3. **Natural disasters.** This category includes fires, earthquakes, floods, and power failures.

Q16. State the two different techniques used for database recovery.

The database recovery process involves bringing the database to a consistent state after a failure. Transaction recovery procedures generally make use of deferred-write and write-through techniques.

When the recovery procedure uses a **deferred-write technique** (also called a **deferred update**), the transaction operations do not immediately update the physical database. Instead, only the transaction log is updated. The database is physically updated only after the transaction reaches its commit point, using information from the transaction log. If the transaction aborts before it reaches its commit point, no changes (no ROLLBACK or undo) need to be made to the database because the database was never updated. The recovery process for all started and committed transactions (before the failure) follows these steps:

1. Identify the last checkpoint in the transaction log. This is the last time transaction data was physically saved to disk.

2. For a transaction that started and was committed before the last checkpoint, nothing needs to be done because the data are already saved.

3. For a transaction that performed a commit operation after the last checkpoint, the DBMS uses the transaction log records to redo the transaction and to update the database, using the "after" values in the transaction log. The changes are made in ascending order, from oldest to newest.

4. For any transaction that had a ROLLBACK operation after the last checkpoint or that was left active (with neither a COMMIT nor a ROLLBACK) before the failure occurred, nothing needs to be done because the database was never updated.

When the recovery procedure uses a **write-through technique** (also called an **immediate update**), the database is immediately updated by transaction operations during the transaction's execution, even before the transaction reaches its commit point. If the transaction aborts before it reaches its commit point, a ROLLBACK or undo operation needs to be done to restore the database to a consistent state. In that case, the ROLLBACK operation will use the transaction log "before" values. The recovery process follows these steps:

1. Identify the last checkpoint in the transaction log. This is the last time transaction data were physically saved to disk.

2. For a transaction that started and was committed before the last checkpoint, nothing needs to be done because the data are already saved.

3. For a transaction that was committed after the last checkpoint, the DBMS redoes the transaction, using the "after" values of the transaction log. Changes are applied in ascending order, from oldest to newest.

4. For any transaction that had a ROLLBACK operation after the last checkpoint or that was left active (with neither a COMMIT nor a ROLLBACK) before the failure occurred, the DBMS uses the transaction log records to ROLLBACK or undo the operations, using the "before" values in the transaction log. Changes are applied in reverse order, from newest to oldest.