UNIT – V

Transaction Processing: Transaction concepts, transaction states, ACID properties, implementation of atomicity and durability, implementation of isolation
Concurrency control: concurrent executions, serializability, testing for serializability, recoverability, concurrency control protocols, pessimistic and optimistic concurrency control schemes, log based recovery, recovery with concurrent transactions, advanced recovery techniques

TRANSACTION CONCEPTS

- The transaction is a set of logically related operation. It contains a group of tasks.
- A transaction is an action or series of actions. It is performed by a single user to perform operations for accessing the contents of the database.

Example: Suppose an employee of bank transfers Rs 800 from X's account to Y's account. This small transaction contains several low-level tasks:

X's Account

1. Open_Account(X)
2. Old_Balance = X.balance
3. New_Balance = Old_Balance - 800
4. X.balance = New_Balance
5. Close_Account(X)

Y's Account

1. Open_Account(Y)
2. Old_Balance = Y.balance
3. New_Balance = Old_Balance + 800
4. Y.balance = New_Balance
5. Close_Account(Y)

Operations of Transaction:

Following are the main operations of transaction:

Read(X): Read operation is used to read the value of X from the database and stores it in a buffer in main memory.

Write(X): Write operation is used to write the value back to the database from the buffer.

Let's take an example to debit transaction from an account which consists of following operations:

1. R(X);
2. X = X - 500;
3. W(X);

Let's assume the value of X before starting of the transaction is 4000.

- The first operation reads X's value from database and stores it in a buffer.
- The second operation will decrease the value of X by 500. So buffer will contain 3500.
- The third operation will write the buffer's value to the database. So X's final value will be 3500.

But it may be possible that because of the failure of hardware, software or power, etc. that transaction may fail before finished all the operations in the set.

For example: If in the above transaction, the debit transaction fails after executing operation 2 then X's value will remain 4000 in the database which is not acceptable by the bank.

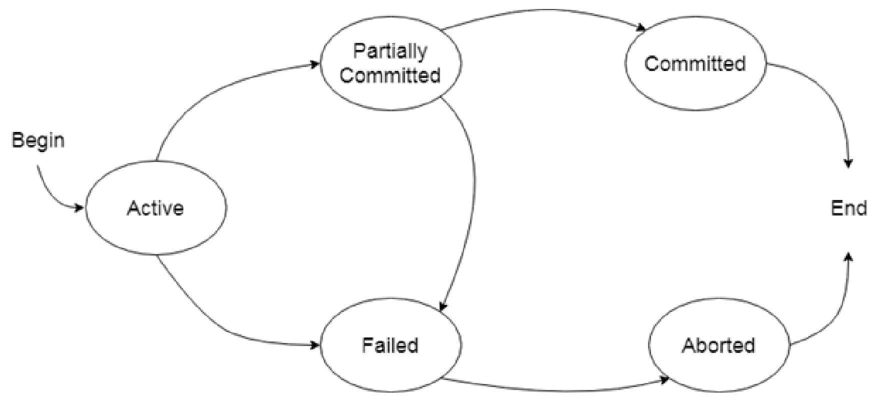
To solve this problem, we have two important operations:

Commit: It is used to save the work done permanently.

Rollback: It is used to undo the work done.

TRANSCTION STATES

In a database, the transaction can be in one of the following states -



Active state

- The active state is the first state of every transaction. In this state, the transaction is being executed.
- For example: Insertion or deletion or updating a record is done here. But all the records are still not saved to the database.

Partially committed

- In the partially committed state, a transaction executes its final operation, but the data is still not saved to the database.
- In the total mark calculation example, a final display of the total marks step is executed in this state.

Committed
A transaction is said to be in a committed state if it executes all its operations successfully. In this state, all the effects are now permanently saved on the database system.

Failed state

- If any of the checks made by the database recovery system fails, then the transaction is said to be in the failed state.
- In the example of total mark calculation, if the database is not able to fire a query to fetch the marks, then the transaction will fail to execute.

Aborted

- If any of the checks fail and the transaction has reached a failed state then the database recovery system will make sure that the database is in its previous consistent state. If not then it will abort or roll back the transaction to bring the database into a consistent state.
- If the transaction fails in the middle of the transaction then before executing the transaction, all the executed transactions are rolled back to its consistent state.
- After aborting the transaction, the database recovery module will select one of the two operations:

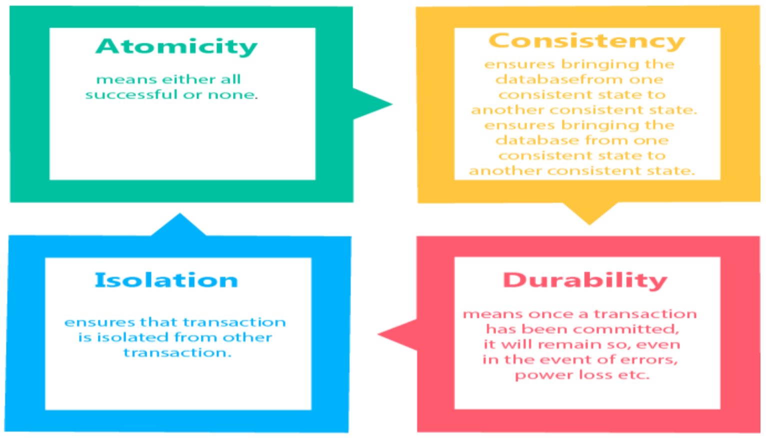
    - Re-start the transaction
    - Kill the transaction

ACID PROPERTIES

The transaction has the four properties. These are used to maintain consistency in a database, before and after the transaction.

Property of Transaction

1. Atomicity
2. Consistency
3. Isolation
4. Durability


Atomicity
means either all successful or none.
Consistency
ensures bringing the databasefrom one consistent state to another consistent state. ensures bringing the database from one consistent state to another consistent state.
Isolation
ensures that transaction is isolated from other transaction.
Durability
means once a transaction has been committed, it will remain so, even in the event of errors, power loss etc.

Atomicity

- It states that all operations of the transaction take place at once if not, the transaction is aborted.
- There is no midway, i.e., the transaction cannot occur partially. Each transaction is treated as one unit and either run to completion or is not executed at all.

Atomicity involves the following two operations:

Abort: If a transaction aborts then all the changes made are not visible.
Commit: If a transaction commits then all the changes made are visible.
Example: Let's assume that following transaction T consisting of T1 and T2. A consists of Rs 600 and B consists of Rs 300. Transfer Rs 100 from account A to account B.

T1	T2

Read(A)
A:= A-100
Write(A)
	Read(B)
	Y:= Y+100
	Write(B)

After completion of the transaction, A consists of Rs 500 and B consists of Rs 400.
If the transaction T fails after the completion of transaction T1 but before completion of transaction T2, then the amount will be deducted from A but not added to B. This shows the inconsistent database state. In order to ensure correctness of database state, the transaction must be executed in entirety.

Consistency

- The integrity constraints are maintained so that the database is consistent before and after the transaction.
- The execution of a transaction will leave a database in either its prior stable state or a new stable state.
- The consistent property of database states that every transaction sees a consistent database instance.

- The transaction is used to transform the database from one consistent state to another consistent state.

For example: The total amount must be maintained before or after the transaction.

1. Total before T occurs = 600+300=900
2. Total after T occurs= 500+400=900
   Therefore, the database is consistent. In the case when T1 is completed but T2 fails, then inconsistency will occur.

Isolation

- It shows that the data which is used at the time of execution of a transaction cannot be used by the second transaction until the first one is completed.
- In isolation, if the transaction T1 is being executed and using the data item X, then that data item can't be accessed by any other transaction T2 until the transaction T1 ends.
- The concurrency control subsystem of the DBMS enforced the isolation property.

Durability

- The durability property is used to indicate the performance of the database's consistent state. It states that the transaction made the permanent changes.
- They cannot be lost by the erroneous operation of a faulty transaction or by the system failure. When a transaction is completed, then the database reaches a state known as the consistent state. That consistent state cannot be lost, even in the event of a system's failure.
- The recovery subsystem of the DBMS has the responsibility of Durability property.

IMPLEMENTATION OF ATOMICITY AND DURABILITY

Atomicity and durability are two important concepts in database management systems (DBMS) that ensure the consistency and reliability of data.

Atomicity:

One of the key characteristics of transactions in database management systems (DBMS) is atomicity, which guarantees that every operation within a transaction is handled as a single, indivisible unit of work.A key characteristic of transactions in database management systems is atomicity (DBMS). It makes sure that every action taken as part of a transaction is handled as a single, indivisible item of labor that can either be completed in full or not at all.

Even in the case of mistakes, failures, or crashes, atomicity ensures that the database maintains consistency. The following are some of the reasons why atomicity is essential in DBMS:

- **Consistency:** Atomicity ensures that the database remains in a consistent state at all times. All changes made by a transaction are rolled back if it is interrupted or fails for any other reason, returning the database to its initial state. By doing this, the database's consistency and data integrity are maintained.

- **Recovery:** Atomicity guarantees that, in the event of a system failure or crash, the database can be restored to a consistent state. All changes made by a transaction are undone if it is interrupted or fails, and the database is then reset to its initial state using the undo log. This guarantees that, even in the event of failure, the database may be restored to a consistent condition.
- **Concurrency:** Atomicity makes assurance that transactions can run simultaneously without affecting one another. Each transaction is carried out independently of the others, and its modifications are kept separate. This guarantees that numerous users can access the database concurrently without resulting in conflicts or inconsistent data.
- **Reliability:** Even in the face of mistakes or failures, atomicity makes the guarantee that the database is trustworthy. By ensuring that transactions are atomic, the database remains consistent and reliable, even in the event of system failures, crashes, or errors.

- **Undo Log:** An undo log is a mechanism used to keep track of the changes made by a transaction before it is committed to the database. If a transaction fails, the undo log is used to undo the changes made by the transaction, effectively rolling back the transaction. By doing this, the database is guaranteed to remain in a consistent condition.
- **Redo Log:** A redo log is a mechanism used to keep track of the changes made by a transaction after it is committed to the database. If a system failure occurs after a transaction is committed but before its changes are written to disk, the redo log can be used to redo the changes and ensure that the database is consistent.
- **Two-Phase Commit:** Two-phase commit is a protocol used to ensure that all nodes in a distributed system commit or abort a transaction together. This ensures that the transaction is executed atomically across all nodes and that the database remains consistent across the entire system.
- **Locking:** Locking is a mechanism used to prevent multiple transactions from accessing the same data concurrently. By ensuring that only one transaction can edit a specific piece of data at once, locking helps to avoid conflicts and maintain the consistency of the database.

Durability:

One of the key characteristics of transactions in database management systems (DBMS) is durability, which guarantees that changes made by a transaction once it has been committed are permanently kept in the database and will not be lost even in the case of a system failure or catastrophe.

Importance:
Durability is a critical property of transactions in database management systems (DBMS) that ensures that once a transaction is committed, its changes are permanently stored in the database and will not be lost, even in the event of a system failure or crash. The following are some of the reasons why durability is essential in DBMS:

- **Data Integrity:** Durability ensures that the data in the database remains consistent and accurate, even in the event of a system failure or crash. It guarantees that committed transactions are durable and will be recovered without data loss or corruption.
- **Reliability:** Durability guarantees that the database will continue to be dependable despite faults or failures. In the event of system problems, crashes, or failures, the database is kept consistent and trustworthy by making sure that committed transactions are durable.

- **Recovery:** Durability guarantees that, in the event of a system failure or crash, the database can be restored to a consistent state. The database can be restored to a consistent state if a committed transaction is lost due to a system failure or crash since it can be recovered from the redo log or other backup storage.
- **Availability:** Durability ensures that the data in the database is always available for access by users, even in the event of a system failure or crash. It ensures that committed transactions are always retained in the database and are not lost in the event of a system crash.

The implementation of durability in DBMS involves several techniques to ensure that committed changes are durable and can be recovered in the event of failure.

- **Write-Ahead Logging:** Write-ahead logging is a mechanism used to ensure that changes made by a transaction are recorded in the redo log before they are written to the database. This makes sure that the changes are permanent and that they can be restored from the redo log in the event of a system failure.
- **Checkpointing:** Checkpointing is a technique used to periodically write the database state to disk to ensure that changes made by committed transactions are permanently stored. Checkpointing aids in minimizing the amount of work required for database recovery.
- **Redundant storage:** Redundant storage is a technique used to store multiple copies of the database or its parts, such as the redo log, on separate disks or systems. This ensures that even in the event of a disk or system failure, the data can be recovered from the redundant storage.
- **RAID:** In order to increase performance and reliability, a technology called RAID (Redundant Array of Inexpensive Disks) is used to integrate several drives into a single logical unit. RAID can be used to implement redundancy and ensure that data is durable even in the event of a disk failure.

- **Transactions:** Transactions are used to group related operations that need to be executed atomically. They are either committed, in which case all their changes become permanent, or rolled back, in which case none of their changes are made permanent.
- **Logging:** Logging is a technique that involves recording all changes made to the database in a separate file called a log. The log is used to recover the database in case of a failure. Write-ahead logging is a common technique that guarantees that data is written to the log before it is written to the database.
- **Shadow Paging:** Shadow paging is a technique that involves making a copy of the database before any changes are made. The copy is used to provide a consistent view of the database in case of failure. The modifications are made to the original database after a transaction has been committed.
- **Backup and Recovery:** In order to guarantee that the database can be recovered to a consistent state in the event of a failure, backup and recovery procedures are used. This involves making regular backups of the database and keeping track of changes made to the database since the last backup.

IMPLEMENTATION OF ISOLATION

Isolation is one of the core ACID properties of a database transaction, ensuring that the operations of one transaction remain hidden from other transactions until completion. It means that no two transactions should interfere with each other and affect the other's intermediate state.

Isolation Levels

Isolation levels defines the degree to which a transaction must be isolated from the data modifications made by any other transaction in the database system. There are four levels of transaction isolation defined by SQL -

1. Serializable

- The highest isolation level.
- Guarantees full serializability and ensures complete isolation of transaction operations.

2. Repeatable Read

- This is the most restrictive isolation level.
- The transaction holds read locks on all rows it references.
- It holds write locks on all rows it inserts, updates, or deletes.
- Since other transaction cannot read, update or delete these rows, it avoids non repeatable read.

3. Read Committed

- This isolation level allows only committed data to be read.
- Thus it does not allows dirty read (i.e. one transaction reading of data immediately after written by another transaction).
- The transaction hold a read or write lock on the current row, and thus prevent other rows from reading, updating or deleting it.

4. Read Uncommitted

- It is lowest isolation level.
- In this level, one transaction may read not yet committed changes made by other transaction.
- This level allows dirty reads.

The proper isolation level or concurrency control mechanism to use depends on the specific requirements of a system and its workload. Some systems may prioritize high throughput and can tolerate lower isolation levels, while others might require strict consistency and higher isolation.

Isolation level	**Dirty Read**	**Unrepeatable Read**
Serializable	NO	NO
Repeatable Read	NO	NO
Read Committed	NO	Maybe
Read Uncommitted	Maybe	Maybe

CONCURRENT EXECUTIONS

- In a multi-user system, multiple users can access and use the same database at one time, which is known as the concurrent execution of the database. It means that the same database is executed simultaneously on a multi-user system by different users.
- While working on the database transactions, there occurs the requirement of using the database by multiple users for performing different operations, and in that case, concurrent execution of the database is performed.
- Executing a single transaction at a time will increase the waiting time of the other transactions which may result in delay in the overall execution. Hence for increasing the overall throughput and efficiency of the system, several transactions are executed.
- 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

Implementing isolation typically involves concurrency control mechanisms. Here are common mechanisms used:

1. Locking Mechanisms

Locking ensures exclusive access to a data item for a transaction. This means that while one transaction holds a lock on a data item, no other transaction can access that item.

- **Shared Lock (S-lock):** Allows a transaction to read an item but not write to it.
- **Exclusive Lock (X-lock):** Allows a transaction to read and write an item. No other transaction can read or write until the lock is released.
- **Two-phase Locking (2PL):** This protocol ensures that a transaction acquires all the locks before it releases any. This results in a growing phase (acquiring locks and not releasing any) and a shrinking phase (releasing locks and not acquiring any).

2. Timestamp-based Protocols

Every transaction is assigned a unique timestamp when it starts. This timestamp determines the order of transactions. Transactions can only access the database if they respect the timestamp order, ensuring older transactions get priority.

Concurrent execution refers to the **simultaneous execution** of more than one transaction. This is a common scenario in multi-user database environments where many users or applications might be accessing or modifying the database at the same time. Concurrent execution is crucial for achieving high throughput and efficient resource utilization. However, it introduces the potential for conflicts and data inconsistencies.

Advantages of Concurrent Execution

1. **Increased System Throughput:** Multiple transactions can be in progress at the same time, but at different stages
2. **Maximized Processor Utilization:** If one transaction is waiting for I/O operations, another transaction can utilize the processor.
3. **Decreased Wait Time:** Transactions no longer have to wait for other long transactions to complete.
4. **Improved Transaction Response Time:** Transactions get processed faster because they can be executed in parallel.

Potential Problems with Concurrent Execution

1. Lost Update Problem (Write-Write conflict):

One transaction's updates could be overwritten by another.

Examples:

T1	**T2**
Read(A)	
A = A+50	
	Read(A)
	A = A+100
Write(A)	
	Write(A)

Result: T1's updates are lost.

2.Temporary Inconsistency or Dirty Read Problem (Write-Read conflict):

One transaction might read an inconsistent state of data that's being updated by another.

Examples:

T1	**T2**
Read(A)	
A = A+50	
Write(A)	
	Read(A)
	A = A+100
	Write(A)
Read(A)(rollbacks)	

	commit

Result: T2 has a "dirty" value, that was never committed in T1 and doesn't actually exist in the database.

3. Unrepeatable Read Problem (Read-Write conflict):

when a single transaction reads the same row multiple times and observes different values each time. This occurs because another concurrent transaction has modified the row between the two reads.

Examples:

T1	**T2**
Read(A)	
	Read(A)
	A = A+100
	Write(A)
Read(A)	

Result: Within the same transaction, T1 has read two different values for the same data item. This inconsistency is the unrepeatable read.

To manage concurrent execution and ensure the consistency and reliability of the database, DBMSs use concurrency control techniques. These typically include locking mechanisms, timestamps, optimistic concurrency control, and serializability checks.

SERIALIZABILITY

Serializability is the property that ensures that the concurrent execution of a set of transactions produces the same result as if these transactions were executed one after the other without any overlapping, i.e., serially.

Schedule is an order of multiple transactions executing in concurrent environment.
Serial Schedule: The schedule in which the transactions execute one after the other is called serial schedule. It is consistent in nature.

For example: Consider following two transactions T1 and T2.

T1	**T2**
Read(A)	
Write(A)	
Read(B)	

Write(B)	
	Read(A)
	Write(A)
	Read(B)
	Write(B)

All the operations of transaction T1 on data items A and then B executes and then in transaction T2 all the operations on data items A and B execute.

Non Serial Schedule: The schedule in which operations present within the transaction are intermixed. This may lead to conflicts in the result or inconsistency in the resultant data.

For example- Consider following two transactions,

T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(B)
Read(A)	
Write(B)	
	Read(B)
	Write(B)

The above transaction is non serial which result in inconsistency or conflicts in the data.

Why is Serializability Important?

In a database system, for performance optimization, multiple transactions often run concurrently. While concurrency improves performance, it can introduce several data inconsistency problems if not managed properly. Serializability ensures that even when transactions are executed concurrently, the database remains consistent, producing a result that's equivalent to a serial execution of these transactions.

TESTING FOR SERIALIZABILITY

Testing for serializability in a DBMS involves verifying if the interleaved execution of transactions maintains the consistency of the database. The most common way to test for serializability is using a precedence graph (also known as a serializability graph or conflict graph).

Types of Serializability

1.Conflict Serializability

2.View Serializability

Conflict Serializability

Conflict serializability is a form of serializability where the order of non-conflicting operations is not significant. It determines if the concurrent execution of several transactions is equivalent to some serial execution of those transactions.

Two operations are said to be in conflict if:

- They belong to different transactions.
- They access the same data item.
- At least one of them is a write operation.

Examples of non-conflicting operations

T1	T2
Read(A)	Read(A)
Read(A)	Read(B)
Write(B)	Read(A)
Read(B)	Write(A)
Write(A)	Write(B)

Examples of conflicting operations

T1	T2
Read(A)	Write(A)
Write(A)	Read(A)
Write(A)	Write(A)

A schedule is conflict serializable if it can be transformed into a serial schedule (i.e., a schedule with no overlapping transactions) by swapping non-conflicting operations. If it is not possible to transform a given schedule to any serial schedule using swaps of non-conflicting operations, then the schedule is not conflict serializable.

To determine if S is conflict serializable:
Precedence Graph (Serialization Graph): Create a graph where:

Nodes represent transactions.

Draw an edge from \(T_i \) to \(T_j \) if an operation in \(T_i \) precedes and conflicts with an operation in \(T_j \).

For the given example:

T1	T2
Read(A)	

	Read(A)
Write(A)	
	Read(B)
	Write(B)

(R1(A) \) conflicts with W1(A), so there's an edge from T1 to T1, but this is ignored because they´re from the same transaction.
R2(A) conflicts with W1(A), so there's an edge from T2 to T1.
No other conflicting pairs.
The graph has nodes T1 and T2 with an edge from T2 to T1. There are no cycles in this graph.

Decision: Since the precedence graph doesn't have any cycles,Cycle is a path using which we can start from one node and reach to the same node. the schedule S is conflict serializable. The equivalent serial schedules, based on the graph, would be T2 followed by T1.

View Serializability

View Serializability is one of the types of serializability in DBMS that ensures the consistency of a database schedule. Unlike conflict serializability, which cares about the order of conflicting operations, view serializability only cares about the final outcome. That is, two schedules are view equivalent if they have:

- **Initial Read:** The same set of initial reads (i.e., a read by a transaction with no preceding write by another transaction on the same data item).
- **Updated Read:** For any other writes on a data item in between, if a transaction \(T_j\) reads the result of a write by transaction \(T_i\) in one schedule, then \(T_j\) should read the result of a write by \(T_i\) in the other schedule as well.
- **Final Write:** The same set of final writes (i.e., a write by a transaction with no subsequent writes by another transaction on the same data item).

example:

Consider two transactions \(T_1\) and \(T_2\):

Schedule 1(S1):

T1	**T2**
Write(A)	
	Read(A)
	Write(B)
Read(B)	
Write(B)	
Commit	Commit

Schedule 2(S2):

T1	T2
	Read(A)
Write(A)	
	Write(A)
Read(B)	
Write(B)	
Commit	Commit

Here,

1. Both S1 and S2 have the same initial read of A by \(T_2\).
2. Both S1 and S2 have the final write of A by \(T_2\).
3. For intermediate writes/reads, in S1, \(T_2\) reads the value of A after \(T_1\) has written to it. Similarly, in S2, \(T_2\) reads A which can be viewed as if it read the value after \(T_1\) (even though in actual sequence \(T_2\) read it before \(T_1\) wrote it). The important aspect is the view or effect is equivalent.
4. B is read and then written by \(T_1\) in both schedules.

   Considering the above conditions, S1 and S2 are view equivalent. Thus, if S1 is serializable, S2 is also view serializable.

Recoverability refers to the ability of a system to restore its state to a point where the integrity of its data is not compromised, especially after a failure or an error.

When multiple transactions are executing concurrently, issues may arise that affect the system's recoverability. The interaction between transactions, if not managed correctly, can result in scenarios where a transaction's effects cannot be undone, which would violate the system's integrity.

RECOVERABILITY

The need for recoverability arises because databases are designed to ensure data reliability and consistency. If a system isn't recoverable:

- The integrity of the data might be compromised.
- Business processes can be adversely affected due to corrupted or inconsistent data.
- The trust of end-users or businesses relying on the database will be diminished.

Levels of Recoverability

1. Recoverable Schedules

A schedule is said to be recoverable if, for any pair of transactions T_i and T_j, if T_j reads a data item previously written by T_i, then T_i must commit before T_j commits. If a transaction fails for any reason and needs to be rolled back, the system can recover without having to rollback other transactions that have read or used data written by the failed transaction.

Example of a Recoverable Schedule

Suppose we have two transactions T_1 and T_2.

T1	**T2**
Write(A)	
	Read(A)
Commit	
	Write(B)
	Commit

In the above schedule, T_2 reads a value written by T_1, but T_1 commits before T_2, making the schedule recoverable.

2. Non-Recoverable Schedules

A schedule is said to be non-recoverable (or irrecoverable) if there exists a pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, but T_i has not committed yet and T_j commits before T_i. If T_i fails and needs to be rolled back after T_j has committed, there's no straightforward way to roll back the effects of T_j, leading to potential data inconsistency.

Example of a Non-Recoverable Schedule

Again, consider two transactions T_1 and T_2.

T1	**T2**
Write(A)	
	Read(A)
	Write(B)
	Commit
Commit	

In this schedule, T_2 reads a value written by T_1 and commits before T_1 does. If T_1 encounters a failure and has to be rolled back after T_2 has committed, we're left in a problematic situation since we cannot easily roll back T_2, making the schedule non-recoverable.

3. Cascading Rollback

A cascading rollback occurs when the rollback of a single transaction causes one or more dependent transactions to be rolled back. This situation can arise when one transaction reads uncommitted changes of another transaction, and then the latter transaction fails and needs to be rolled back. Consequently, any transaction that has read the uncommitted changes of the failed transaction also needs to be rolled back, leading to a cascade effect.

Example of Cascading Rollback

Consider two transactions \(T_1\) and \(T_2\):

T1	**T2**
Write(A)	
	Read(A)
	Write(B)
Abort some failure	
Rollback	
	Rollback

Here, \(T_2\) reads an uncommitted value of A written by \(T_1\). When \(T_1\) fails and is rolled back, \(T_2\) also has to be rolled back, leading to a cascading rollback. This is undesirable because it wastes computational effort and can complicate recovery procedures.

4. Cascadeless Schedules

A schedule is considered cascadeless if transactions only read committed values. This means, in such a schedule, a transaction can read a value written by another transaction only after the latter has committed. Cascadeless schedules prevent cascading rollbacks.

Example of Cascadeless Schedule

Consider two transactions \(T_1\) and \(T_2\):

T1	**T2**
Write(A)	
Commit	
	Read(A)
	Write(B)
	Commit

In this schedule, \(T_2\) reads the value of A only after \(T_1\) has committed. Thus, even if \(T_1\) were to fail before committing (not shown in this schedule), it would not affect \(T_2\). This means there's no risk of cascading rollback in this schedule.

Locks are essential in a database system to ensure:

1. **Consistency:** Without locks, multiple transactions could modify the same data item simultaneously, resulting in an inconsistent state.
2. **Isolation:** Locks ensure that the operations of one transaction are isolated from other transactions, i.e., they are invisible to other transactions until the transaction is committed.
3. **Concurrency:** While ensuring consistency and isolation, locks also allow multiple transactions to be processed simultaneously by the system, optimizing system throughput and overall performance.
4. **Avoiding Conflicts:** Locks help in avoiding data conflicts that might arise due to simultaneous read and write operations by different transactions on the same data item.
5. **Preventing Dirty Reads:** With the help of locks, a transaction is prevented from reading data that hasn't yet been committed by another transaction.

CONCURRENCY CONTROL PROTOCOLS

Lock-Based Protocols

1. Simple Lock Based Protocol

The Simple lock based protocol is a mechanism in which there is exclusive use of locks on the data item for current transaction.

Types of Locks: There are two types of locks used -

Shared Lock (S-lock)

This lock allows a transaction to read a data item. Multiple transactions can hold shared locks on the same data item simultaneously. It is denoted by **Lock-S**. This is also called as **read lock**.

Exclusive Lock (X-lock):

This lock allows a transaction to read and write a data item. If a transaction holds an exclusive lock on an item, no other transaction can hold any kind of lock on the same item. It is denoted as **Lock-X**. This is also called as **write lock**.

T1	**T2**
Lock-S(A)	

Read(A)	
Unlock(A)	Lock-X(A)
	Write(B)
	Read(A)
	Write(A)
	Unlock(A)

The difference between shared lock and exclusive lock

Shared Lock	**Exclusive Lock**
Shared lock is used for when the transaction wants to perform read operation.	Exclusive lock is used when the transaction wants to perform both read and write operation.
Any number of transactions can hold shared lock on an item.	But exclusive lock can be hold by only one transaction.
Using shared lock data item can be viewed.	Using exclusive lock data can be inserted or deleted.

2. Two-Phase Locking (2PL) Protocol

The two-phase locking protocol ensures a transaction gets all the locks it needs before it releases any. The protocol has two phases:

- **Growing Phase:** The transaction may obtain any number of locks but cannot release any.
- **Shrinking Phase:** The transaction may release but cannot obtain any new locks.

The point where the transaction releases its first lock is the end of the growing phase and the beginning of the shrinking phase.

This protocol ensures conflict-serializability and avoids many of the concurrency issues, but it doesn't guarantee the absence of deadlocks.

T1	**T2**
Lock-S(A)	
	Lock-S(A)
Lock-X(B)	
Unlock(A)	

	Lock-X(C)
Unlock(B)	
	Unlock(A)
	Unlock(C)

Pros of Two-Phase Locking (2PL)

- **Ensures Serializability:** 2PL guarantees conflict-serializability, ensuring the consistency of the database.
- **Concurrency:** By allowing multiple transactions to acquire locks and release them, 2PL increases the concurrency level, leading to better system throughput and overall performance.
- **Avoids Cascading Rollbacks:** Since a transaction cannot read a value modified by another uncommitted transaction, cascading rollbacks are avoided, making recovery simpler.

Cons of Two-Phase Locking (2PL)

- **Deadlocks:** The main disadvantage of 2PL is that it can lead to deadlocks, where two or more transactions wait indefinitely for a resource locked by the other.
- **Reduced Concurrency (in certain cases):** Locking can block transactions, which can reduce concurrency. For example, if one transaction holds a lock for a long time, other transactions needing that lock will be blocked.
- **Overhead:** Maintaining locks, especially in systems with a large number of items and transactions, requires overhead. There's a time cost associated with acquiring and releasing locks, and memory overhead for maintaining the lock table.
- **Starvation:** It's possible for some transactions to get repeatedly delayed if other transactions are continually requesting and acquiring locks.

Categories of Two-Phase Locking in DBMS

1. Strict Two-Phase Locking
2. Rigorous Two-Phase Locking
3. Conservative (or Static) Two-Phase Locking:

Strict Two-Phase Locking

- Transactions are not allowed to release any locks until after they commit or abort.

- Ensures serializability and avoids the problem of cascading rollbacks.
- However, it can reduce concurrency.

Pros Strict Two-Phase Locking

- **Serializability:** Ensures serializable schedules, maintaining the consistency of the database.
- **Avoids Cascading Rollbacks:** A transaction cannot read uncommitted data, thus avoiding cascading aborts.
- **Simplicity:** Conceptually straightforward in that a transaction simply holds onto its locks until it's finished.

Cons Strict Two-Phase Locking

- **Reduced Concurrency:** Since transactions hold onto locks until they commit or abort, other transactions might be blocked for longer periods.
- **Potential for Deadlocks:** Holding onto locks can increase the chances of deadlocks.

Rigorous Two-Phase Locking

- A transaction can release a lock after using it, but it cannot commit until all locks have been acquired.
- Like strict 2PL, rigorous 2PL is deadlock-free and ensures serializability.

Pros Rigorous Two-Phase Locking

- **Serializability:** Like strict 2PL, ensures serializable schedules.
- **Avoids Cascading Rollbacks:** Prevents reading of uncommitted data.
- **Improved Concurrency:** By releasing locks before committing, it can potentially allow higher concurrency compared to strict 2PL.

Cons Rigorous Two-Phase Locking

- **Deadlock Possibility:** Still susceptible to deadlocks as transactions might hold some locks while releasing others.

Conservative or Static Two-Phase Locking

- A transaction must request all the locks it will ever need before it begins execution. If any of the requested locks are unavailable, the transaction is delayed until they are all available.
- This approach can avoid deadlocks since transactions only start when all their required locks are available.

Pros Conservative or Static Two-Phase Locking

- **Avoids Deadlocks:** By ensuring that all required locks are acquired before a transaction starts, it inherently avoids the possibility of deadlocks.
- **Serializability:** Ensures serializable schedules.

Cons Conservative or Static Two-Phase Locking

Reduced Concurrency: Transactions might be delayed until all required locks are available, leading to potential inefficiencies.

Timestamp-based protocols are concurrency control mechanisms used in databases to ensure serializability and to avoid conflicts without the need for locking. The main idea behind these protocols is to use a timestamp to determine the order in which transactions should be executed. Each transaction is assigned a unique timestamp when it starts.

Here are the primary rules associated with timestamp-based protocols:

1. Timestamp Assignment: When a transaction \(T \) starts, it is given a timestamp \(TS(T) \). This timestamp can be the system's clock time or a logical counter that increments with each new transaction.

2. Reading/Writing Rules: Timestamp-based protocols use the following rules to determine if a transaction can read or write an item:

Read Rule: If a transaction \(T \) wants to read an item that was last written by transaction \(T' \) with \(TS(T') > TS(T) \), the read is rejected because it's trying to read a value from the future. Otherwise, it can read the item.

Write Rule: If a transaction \(T \) wants to write an item that has been read or written by a transaction \(T' \) with \(TS(T') > TS(T) \), the write is rejected. This avoids overwriting a value that has been read or written by a younger transaction.

3. Handling Violations: When a transaction's read or write operation violates the rules, the transaction can be rolled back and restarted or aborted, depending on the specific protocol in use.

Let's look at examples to better understand these rules:

Example on Timestamp-based protocols

Suppose two transactions \(T1 \) and \(T2 \) with timestamps 5 and 10 respectively:

1. \(T1 \) reads item \(A \).
2. \(T2 \) writes item \(A \).

3. \(T1 \) tries to write item \(A \).

According to the write rule, \(T1 \) can't write item \(A \) after \(T2 \) has written it, because \(TS(T2) > TS(T1) \). Thus, \(T1 \)'s write operation will be rejected.

Example-2

Suppose two transactions \(T1 \) and \(T2 \) with timestamps 5 and 10 respectively:

1. \(T2 \) writes item \(A \).
2. \(T1 \) tries to read item \(A \).

   According to the read rule, \(T1 \) can't read item \(A \) after \(T2 \) has written it, as \(TS(T2) > TS(T1) \). So, \(T1 \)'s read operation will be rejected.

Advantages of Timestamp-based Protocols

1. **Deadlock-free:** Since there are no locks involved, there's no chance of deadlocks occurring.
2. **Fairness:** Older transactions have priority over newer ones, ensuring that transactions do not starve and get executed in a fair manner.
3. **Increased Concurrency:** In many situations, timestamp-based protocols can provide better concurrency than lock-based methods.

Disadvantages of Timestamp-based Protocols

1. **Starvation:** If a transaction is continually rolled back due to timestamp rules, it may starve.
2. **Overhead:** Maintaining and comparing timestamps can introduce overhead, especially in systems with a high transaction rate.
3. **Cascading Rollbacks:** A rollback of one transaction might cause rollbacks of other transactions.

One of the most well-known timestamp-based protocols is the **Thomas Write Rule**, which modifies the write rule to allow certain writes that would be rejected under the basic timestamp protocol. The idea is to ignore a write that would have no effect on the outcome, rather than rolling back the transaction. This reduces the number of rollbacks but can result in non-serializable schedules.

In practice, timestamp-based protocols offer an alternative approach to concurrency control, especially useful in systems where locking leads to frequent deadlocks or reduced concurrency. However, careful implementation and tuning are required to handle potential issues like transaction starvation or cascading rollbacks.

Validation Based Protocols in DBMS

Validation-based protocols, also known as Optimistic Concurrency Control (OCC), are a set of techniques that aim to increase system concurrency and performance by assuming that conflicts between transactions will be rare. Unlike other concurrency control methods, which try to prevent conflicts proactively using locks or timestamps, OCC checks for conflicts only at transaction commit time.

Here's how a typical validation-based protocol operates:

1. **Read Phase**
   - The transaction reads from the database but does not write to it.
   - All updates are made to a local copy of the data items.
2. **Validation Phase**
   - Before committing, the system checks to ensure that this transaction's local updates won't cause conflicts with other transactions.
   - The validation can take many forms, depending on the specific protocol.
3. **Write Phase**
   - If the transaction passes validation, its updates are applied to the database.
   - If it doesn't pass validation, the transaction is aborted and restarted.

Validation-based protocols Example

Let's consider a simple scenario with two transactions \(T1 \) and \(T2 \):

- Both transactions read an item \(A \) with a value of 10.
- \(T1 \) updates its local copy of \(A \) to 12.
- \(T2 \) updates its local copy of \(A \) to 15.
- \(T1 \) reaches the validation phase and is validated successfully (because there's no other transaction that has written to \(A \) since \(T1 \) read it).
- \(T1 \) moves to the write phase and updates \(A \) in the database to 12.
- \(T2 \) reaches the validation phase. The system realizes that since \(T2 \) read item \(A \), another transaction (i.e., \(T1 \)) has written to \(A \). Therefore, \(T2 \) fails validation.
- \(T2 \) is aborted and can be restarted.

Advantages of Validation-based protocols

- **High Concurrency:** Since transactions don't acquire locks, more than one transaction can process the same data item concurrently.
- **Deadlock-free:** Absence of locks means there's no deadlock scenario.

Disadvantages of Validation-based protocols

- **Overhead:** The validation process requires additional processing overhead.
- **Aborts:** Transactions might be aborted even if there's no real conflict, leading to wasted processing.

Validation-based protocols work best in scenarios where conflicts are rare. If the system anticipates many conflicts, then the high rate of transaction restarts might offset the advantages of increased concurrency.

Multiple Granularity in DBMS

In the context of database systems, **"granularity"** refers to the size or extent of a data item that can be locked by a transaction. The idea behind multiple granularity is to provide a hierarchical structure that allows locks at various levels, ranging from coarse-grained (like an entire table) to fine-grained (like a single row or even a single field). This hierarchy offers flexibility in achieving the right balance between concurrency and data integrity.

The concept of multiple granularity can be visualized as a tree. Consider a database system where:

- The entire database can be locked.
- Within the database, individual tables can be locked.
- Within a table, individual pages or rows can be locked.
- Even within a row, individual fields can be locked.

Lock Modes in multiple granularity

To ensure the consistency and correctness of a system that allows multiple granularity, it's crucial to introduce the concept of "intention locks." These locks indicate a transaction's intention to acquire a finer-grained lock in the future.

There are three main types of intention locks:

1. Intention Shared (IS): When a Transaction needs S lock on a node "K", the transaction would need to apply IS lock on all the precedent nodes of "K", starting from the root node. So, when a node is found locked in IS mode, it indicates that some of its descendent nodes must be locked in S mode.

Example: Suppose a transaction wants to read a few records from a table but not the whole table. It might set an IS lock on the table, and then set individual S locks on the specific rows it reads.

2. Intention Exclusive (IX): When a Transaction needs X lock on a node "K", the transation would need apply IX lock on all the precedent nodes of "K", starting from the root node. So, when a node is found locked in IX mode, it indicates that some of its descendent nodes must be locked in X mode.

Example: If a transaction aims to update certain records within a table, it may set an IX lock on the table and subsequently set X locks on specific rows it updates.

3. Shared Intention Exclusive (SIX): When a node is locked in SIX mode; it indicates that the node is explicitly locked in S mode and Ix mode. So, the entire tree rooted by that node is locked in S mode and some nodes in that are locked in X mode. This mode is compatible only with IS mode.

Example: Suppose a transaction wants to read an entire table but also update certain rows. It would set a SIX lock on the table. This tells other transactions they can read the table but cannot update it until the SIX lock is released. Meanwhile, the original transaction can set X locks on specific rows it wishes to update.

Compatibility Matrix with Lock Modes in multiple granularity

A compatibility matrix defines which types of locks can be held simultaneously on a database object. Here's a simplified matrix:

	NL	IS	IX	S	SIX	X
NL	✓	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	✓	✗
IX	✓	✓	✓	✗	✗	✗
S	✓	✓	✗	✓	✗	✗
SIX	✓	✓	✗	✗	✗	✗
X	✓	✗	✗	✗	✗	✗

(NL = No Lock, S = Shared, X = Exclusive)

Benefits of using multiple granularity

- **Flexibility:** Offers flexibility to transactions in deciding the appropriate level of locking, which can lead to improved concurrency.
- **Performance:** Reduces contention by allowing transactions to lock only those parts of the data that they need.

Challenges in multiple granularity

- **Complexity:** Managing multiple granularity adds complexity to the lock management system.
- **Overhead:** The lock manager needs to handle not just individual locks but also the hierarchy and compatibility of these locks.

PESSIMISTIC CONCURRENCY CONTROL

With pessimistic concurrency control, the DBMS assumes that conflicts between transactions are likely to occur. It is pessimistic – that is, it assumes that if something can go wrong, it will go wrong. This pessimism prevents conflicts from occurring by blocking them before they get a chance to start.

To prevent these conflicts, it *locks* the data that a transaction is using until the transaction is completed. This approach is 'pessimistic' because it assumes the worst-case scenario – that every transaction might lead to a *conflict*. The data is therefore locked in order to prevent conflicts from happening.I've mentioned two technical terms here that need clarification: *locks* and *conflict*.

Pessimistic concurrency control is very safe because it prevents conflicts from occurring by blocking them before they get a chance to start. A write lock on a database item prevents other transactions from reading or updating that item while that lock is held, similar to to how a library stops more than one person from trying to borrow the same physical book at the same time.

A read lock on a database item allows other transactions to also obtain a read lock for that item, but prevents transactions from updating that item. This is analogous to borrowing an e-book, where multiple people can borrow the same e-book at the same time, but can’t make any updates to it.

A Simple Real-World Example of Pessimistic Concurrency Control in Action

Assume we have a table named Accounts with the following columns: AccountID and Balance.

AccountID	Balance
12345	$1500

Database columns for AccountID and Balance

Two transactions, T1 and T2, intend to update the balance of account 12345. T1 wants to withdraw $300, and T2 wants to deposit $400. At the end of these two transactions, the account balance should read $1600.

Here are the steps of how this will work using write locks:

1. Start of T1 (Withdrawal): T1 requests to update the balance of AccountID 12345. The database system places an exclusive write lock on the row for AccountID 12345, preventing other transactions from reading or writing to this row until T1 is completed. T1 reads the balance ($1500).
2. T1 Processing: T1 calculates the new balance as $1200 ($1500 - $300).
3. Commit T1: T1 writes the new balance ($1200) back to the database. Upon successful commit, T1 releases the exclusive lock on AccountID 12345.
4. Start of T2 (Deposit) After T1 Completes: Now that T1 has completed and the lock is released, T2 can start. T2 attempts to read and update the balance for AccountID 12345. The database

system places an exclusive lock on the row for AccountID 12345 for T2, ensuring no other transactions can interfere. T2 reads the updated balance ($1200).

5. T2 Processing: T2 calculates the new balance as $1600 ($1200 + $400).
6. Commit T2: T2 writes the new balance ($1600) back to the database. Upon successful commit, T2 releases the exclusive lock on AccountID 12345.
7. Result: The Accounts table is updated using locks After T1: $1200 After T2: $1600
   Without a write lock in this example, T1 and T2 could read the original balance of $1500 at the same time. So, instead of a balance of $1200 after T1 has committed, T2 still reads the original balance of $1500 and adds $400. This would cause the final balance to be $1500 + $400 = $1900 (instead of $1600).

Absence of locking has created free money, which is never a bad thing for a customer. But, if money can be conjured out of thin air because of these conflicts, it can also vanish, and accidentally shrinking bank balances are a quick way to make customers unhappy.

How Pessimistic Concurrency Controls Guarantee the Read Committed Isolation Level

So, how exactly does pessimistic concurrency control work in ensuring the isolation guarantee, that is the “I” in ACID? The implementation details can vary across different DBMS. But the explanation here shows the general approach.

the read committed isolation level prevents dirty writes and dirty reads.

Preventing Dirty Writes

Overwriting data that has already been written by another transaction but not yet committed is called a dirty write. A common approach to preventing dirty writes is to use pessimistic concurrency control. For example, by using a write lock at the row level.

When a transaction wants to modify a row, it acquires a lock on that row and holds it until the transaction is complete. Recall that write locks can only be held by a single transaction. This prevents another transaction from acquiring a lock to modify that row.

Preventing Dirty Reads

Reading data from another transaction that has not yet been committed is called a dirty read. Dirty reads are prevented using either a read or write lock. Once a transaction acquires a read lock on a database item, it will prevent updates to that item.

But what happens if you are trying to read something that is already being updated but the transaction has not yet committed? In this instance, the write lock saves the day again.

Since write locks are exclusive (can’t be shared with other transactions), any transaction wanting to read the same database item will have to wait until the transaction with the write lock is committed (or aborted, if it fails). This prevents other transactions from reading uncommitted changes.

OPTIMISTIC CONCURRENCY CONTROL

With optimistic concurrency control, transactions do not obtain locks on data when they read or write. The "Optimistic" in the name comes from assuming that conflicts are unlikely to occur, so locks are not needed. If something does go wrong though, conflicts will still be prevented and everything will be OK.

Unlike pessimistic concurrency control – which prevents conflicts from occurring by blocking them before they get a chance to start – optimistic concurrency control checks for conflicts at the end of a transaction. With optimistic concurrency control, multiple transactions can read or update the same database item without acquiring locks.

A Simple Real-World Example of Optimistic Concurrency Control in Action

Assume we have a table named Accounts with the following columns: AccountID, Balance, VersionNumber, and Timestamp.

AccountID	Balance	VersionNumber	Timestamp
12345	$1000	1	2024-01 10:00:00

Table showing AccountID, Balance, VersionNumber, and Timestamp columns

Two transactions, T1 and T2, intend to update the balance of account 12345 at the same time. T1 wants to withdraw $200, and T2 wants to deposit $300. At the end of these two transactions, the account balance should read $1100

Here are the steps of how this will work:

1. Start of Transactions: T1 reads the balance, version number, and timestamp for AccountID 12345. Simultaneously, T2 reads the same row with the same balance, version number, and timestamp.
2. Processing: T1 calculates the new balance as $800 ($1000 - $200) but does not write it back immediately. T2 calculates the new balance as $1300 ($1000 + $300) but also waits to commit.
3. Attempt to Commit T1: Before committing, T1 checks the current VersionNumber and Timestamp of AccountID 12345 in the database. Since no other transaction has modified the row, T1 updates the balance to $800, increments the VersionNumber to 2, updates the Timestamp, and commits successfully.
4. Attempt to Commit T2: T2 attempts to commit by first verifying the VersionNumber and Timestamp. T2 finds that the VersionNumber and Timestamp have changed (now VersionNumber is 2, and Timestamp is updated), indicating another transaction (T1) has updated the row. Since the version number and timestamp have changed, T2 realises there was a conflict.

5. Resolution for T2: T2 must restart its transaction. It re-reads the updated balance of $800, the new VersionNumber 2, and the updated Timestamp. T2 recalculates the new balance as $1100 ($800 + $300), updates the VersionNumber to 3, updates the Timestamp, and commits successfully.
   Result: The Accounts table is updated sequentially and safely without any locks: After T1: $800, VersionNumber: 2. After T2: $1100, VersionNumber: 3.

In database management systems (DBMS), ensuring the consistency and durability of data is crucial. One of the key mechanisms used to achieve this is log-based recovery. In this article, we'll explore what log-based recovery is, how it works, and its importance in maintaining the integrity of databases.

LOG-BASED RECOVERY

Log-based recovery is a technique used in DBMS to recover the database to a consistent state in the event of a system failure or crash. It relies on maintaining a transaction log, which is a record of all the transactions that have been executed on the database. The log contains information about the operations performed by each transaction, such as updates, inserts, and deletes, along with the old and new values of the affected data items.

How does Log-based Recovery Work?

Log-based recovery works by using the transaction log to redo or undo transactions as necessary to bring the database back to a consistent state. There are two main phases involved in log-based recovery: redo and undo.

What is Redo Phase?
During the redo phase, the DBMS replays the operations recorded in the transaction log that were not completed before the crash. This ensures that all committed changes are applied to the database, even if they were not physically written to disk before the crash.

What is Undo Phase?
In the undo phase, the DBMS identifies transactions that were active but not committed at the time of the crash. It then reverses or undoes the operations of these transactions to restore the database to its state before these transactions began.

Importance of Log-based Recovery
Log-based recovery is essential for maintaining the consistency and durability of a database in the face of system failures. Here are some key reasons why log-based recovery is important.

- **Atomicity and Durability:** Log-based recovery ensures that transactions are either fully completed or fully undone, preserving the atomicity property of transactions. It also ensures that committed changes are durable and not lost due to a system failure.

- **Recovery from Crashes:** Log-based recovery allows the DBMS to recover from crashes quickly and efficiently, minimizing downtime and ensuring that the database remains available and reliable.
- **Point-in-time Recovery:** Log-based recovery enables point-in-time recovery, allowing the database to be restored to a specific point in time before the crash occurred. This can be useful for recovering from user errors or other types of data corruption.
- **Transaction Rollback:** Log-based recovery enables the rollback of transactions that have not been committed, ensuring that no partial changes are left in the database after a crash.

Log-based recovery is a critical component of modern database management systems, ensuring the consistency, durability, and recoverability of data in the face of system failures. By maintaining a transaction log and using it to redo and undo transactions as necessary, DBMS can recover from crashes and ensure that data remains safe and reliable.

RECOVERY WITH CONCURRENT TRANSACTIONS

Concurrency control means that multiple transactions can be executed at the same time and then the interleaved logs occur. But there may be changes in transaction results so maintain the order of execution of those transactions.

During recovery, it would be very difficult for the recovery system to backtrack all the logs and then start recovering.

Recovery with concurrent transactions can be done in the following four ways.

1. Interaction with concurrency control
2. Transaction rollback
3. Checkpoints
4. Restart recovery

Interaction with Concurrency Control:

This pertains to how the recovery system interacts with the concurrency control component of the DBMS.

Recovery must be aware of concurrency control mechanisms such as locks to ensure that, during recovery, operations are redone or undone in a manner consistent with the original schedule of transactions.

For example, if strict two-phase locking is used, it can simplify the recovery process. Under this protocol, once a transaction releases its first lock, it cannot acquire any new locks. This ensures that if a transaction commits, its effects are permanent and won't be overridden by any other transaction that was running concurrently.

Transaction Rollback

Sometimes, instead of recovering the whole system, it's more efficient to just rollback a particular transaction that has caused inconsistency or when a deadlock occurs.

When errors are detected during transaction execution (like constraint violations) or if a user issues a rollback command, the system uses the logs to undo the actions of that specific transaction, ensuring the database remains consistent.

The system keeps track of all the operations performed by a transaction. If a rollback is necessary, these operations are reversed using the log records.

Checkpoints

Checkpointing is a technique where the DBMS periodically takes a snapshot of the current state of the database and writes all changes to the disk. This reduces the amount of work during recovery.

By creating checkpoints, recovery operations don't need to start from the very beginning. Instead, they can begin from the most recent checkpoint, thereby considerably speeding up the recovery process.

During the checkpoint, ongoing transactions might be temporarily suspended, or their logs might be force-written to the stable storage, depending on the implementation.

Restart Recovery

In the case of a system crash, the recovery manager uses the logs to restore the database to the most recent consistent state. This process is called restart recovery.

Initially, an analysis phase determines the state of all transactions at the time of the crash. Committed transactions and those that had not started are ignored.

The redo phase then ensures that all logged updates of committed transactions are reflected in the database. Lastly, the undo phase rolls back the transactions that were active during the crash to ensure atomicity.

The presence of checkpoints can make the restart recovery process faster since the system can start the redo phase from the most recent checkpoint rather than from the beginning of the log.

Advanced recovery techniques

The database recovery techniques in DBMS are used to recover the data at times of system failure. The recovery techniques in DBMS maintain the properties of atomicity and durability of the database. A system is not called durable if it fails during a transaction and loses all its data and a system is not called atomic, if the data is in a partial state of update during the transaction. The data recovery techniques in DBMS make sure, that the state of data is preserved to protect the atomic property and the data is always recoverable to protect the durability property. The following techniques are used to recover data in a DBMS,

- Log-based recovery in DBMS.
- Recovery through Deferred Update
- Recovery through Immediate Update

A **checkpoint** is made after all the records of a transaction are written to logs to transfer all the logs from the local storage to the permanent storage for future use. Read more on Checkpoint in DBMS.

ADVANCED RECOVERY TECHNIQUES

What is the Conceded Update Method?

In the conceded update method, the updates are not made to the data until the transaction reaches the final phase or at the **commit** operating. After this operation is performed, the data is modified and permanently stored in the main memory. The logs are maintained throughout the operation and are used in case of failure to find the point of failure. This provides us an advantage as even if the system fails before the **commit** stage, the data in the database will not be modified and the status will be managed. If the system fails after the **commit** stage, we can redo the changes to the new stage easily compared to the process involved with the undo operation.

What is the Difference Between a Deferred Update and an Immediate Update?

Deferred updates and immediate updates are database recovery techniques in DBMS that are used to maintain the transaction log files of the DBMS.

In the **Deferred** update, the state of the data in the database is not changed immediately after any transaction is executed, but as soon as the commit has been made, the changes are recorded in the log file, and the state of data is changed.

In the **Immediate** update, at every transaction, the database is updated directly, and a log file is also maintained containing the old and new values.

Deferred Update	Immediate Update
During a transaction, the changes are not applied immediately to the data	An immediate change is made in the database as soon as the transaction occurs.
The log file holds the changes that are going to be applied.	The log file holds the changes along with the new and old values
Buffering and Caching are used in this technique	Shadow paging is used in this technique
More time is required to recover the data when a system failure occurs	A large number of I/O operations are performed to manage the logs during the transaction
If a rollback is made, the log files are destroyed and no change is made to the database	If a rollback is made, the old state of the data is restored with the records in the log file

What are the Backup Techniques?

A backup is a copy of the current state of the database that is stored in another location. This backup is useful in cases when the system is destroyed by natural disasters or physical damage. These backups can be used to recover the database to the state at which the backup was made. Different methods are being used in backup which is as follows,

- An **Immediate backup** are copies that are kept in devices like hard disks or any other dives. When a disk crashes or any technical fault occurs we can use these data to recover the data.
- An **Archival backup** is a copy of the database kept in cloud environments or large storage systems in another region. These are used to recover the data when the system is affected by a natural disaster.

What are Transaction Logs?

The transaction logs are used to keep track of all the transactions that have made an update to the data in the database. The following steps are followed to recover the data using transaction logs,

- The recovery manager searches through all the log files and finds the transactions that have a **start_transaction** stage and doesn't have the **commit** stage.
- The transactions that are of the above case are rolled back to the old state with the help of the logs using the rollback command
- The transactions that have a **commit** command will have made changes to the database and these changes are recorded in the logs. These changes will also be reverted using the undo operation.

What is Shadow Paging?

- In shadow paging, a database is split into n- multiple pages that represent a fixed-size disk memory.
- Similarly, n shadow pages are also created which are copies of the real database.
- At the beginning of a transaction, the state in the database is copied to the shadow pages.
- The shadow pages will not be modified during the transaction and only the real database will be modified.
- When the transaction reaches the **commit** stage, the changes are made to the shadow pages. The changes are made in a way that if the i-th part of the hard disk has modifications, then the i-th shadow page is also modified.
- If there is a failure of the system, the real pages of the database are compared with the shadow pages, and recovery operations are performed.

In the **Caching/Buffering** method, a collection of buffers called DBMS buffers are present in the logical memory. The buffers are used to store all logs during the process and the update to the main log file is made when the transaction reaches the **commit** stage.

--END--