UNIT - IV

Transaction Concept, Transaction State, Implementation of Atomicity and Durability, Concurrent Executions, Serializability, Recoverability, Implementation of Isolation, Testing for serializability, Lock Based Protocols, Timestamp Based Protocols, Validation- Based Protocols, Multiple Granularity, Recovery and Atomicity, Log–Based Recovery, Recovery with Concurrent Transactions.

Transaction State in DBMS

Each and every transactions in DBMS, pass through several states during their lifespan. These states, known as "**Transaction States**," collectively constitute the "**Transaction Lifecycle**." Here are the main states in the lifecycle of a transaction:

Transactions are a set of operations used to perform a logical set of work. It is the bundle of all the instructions of a logical operation. A transaction usually means that the data in the database has changed. One of the major uses of DBMS is to protect the user's data from system failures. It is done by ensuring that all the data is restored to a consistent state when the computer is restarted after a crash. The transaction is any one execution of the user program in a DBMS. One of the important properties of the transaction is that it contains a finite number of steps. Executing the same program multiple times will generate multiple transactions.

Example: Consider the following example of transaction operations to be performed to withdraw cash from an ATM vestibule.

Steps for ATM Transaction

1. Transaction Start.
2. Insert your ATM card.
3. Select a language for your transaction.
4. Select the Savings Account option.
5. Enter the amount you want to withdraw.

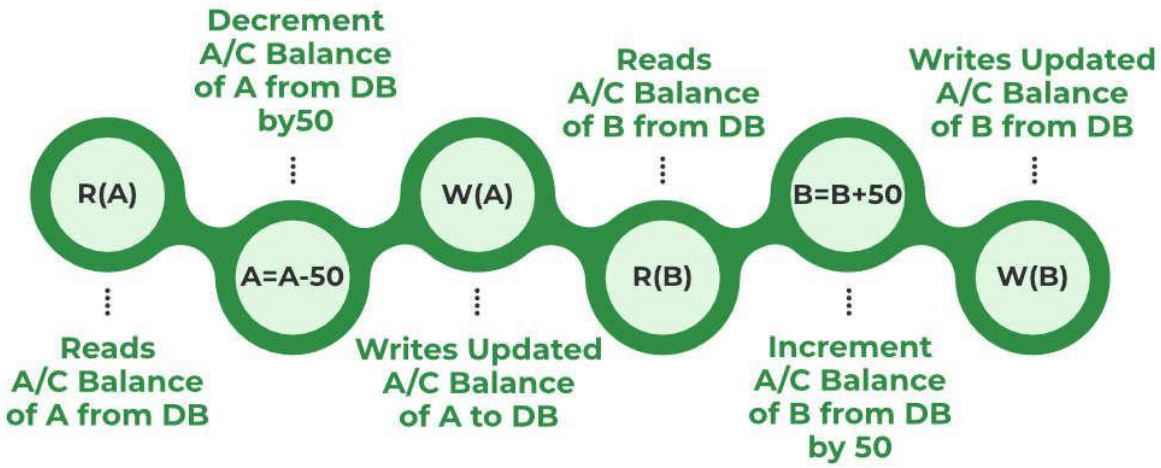
6. Enter your secret pin.
7. Wait for some time for processing.
8. Collect your Cash.
9. Transaction Completed.

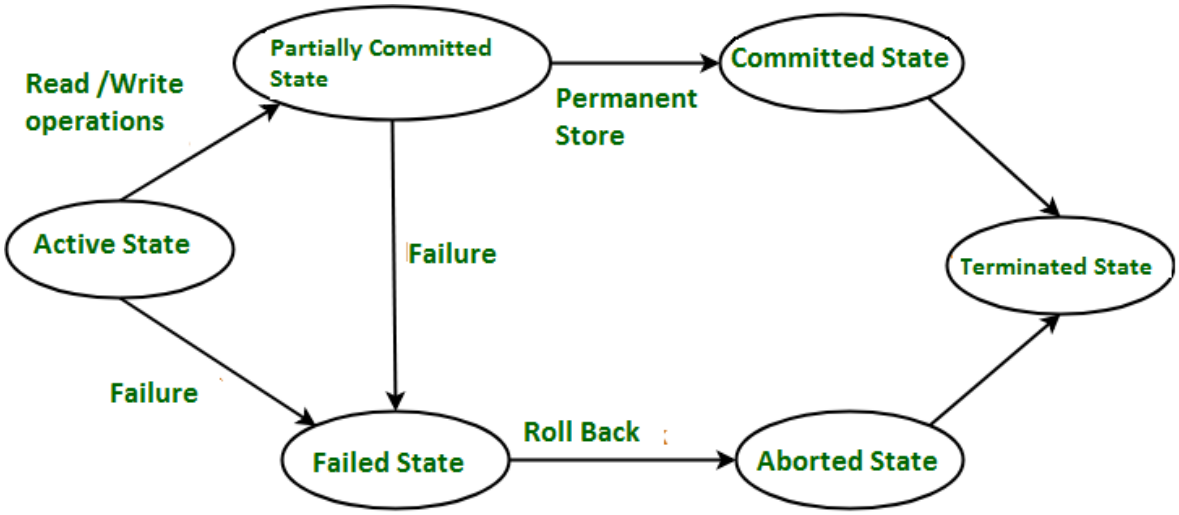
A transaction can include the following basic database access operation.

- **Read/Access data (R):** Accessing the database item from disk (where the database stored data) to memory variable.
- **Write/Change data (W): Write** the data item from the memory variable to the disk.
- **Commit:** Commit is a transaction control language that is used to permanently save the changes done in a transaction

Example: Transfer of 50₹ from Account A to Account B. Initially A= 500₹, B= 800₹. This data is brought to RAM from Hard Disk.

```
R(A) -- 500        // Accessed from RAM.
A = A-50           // Deducting 50₹ from A.
W(A)--450          // Updated in RAM.
R(B) -- 800        // Accessed from RAM.
B=B+50             // 50₹ is added to B's Account.
W(B) --850         // Updated in RAM.
commit             // The data in RAM is taken back to Hard Disk.
```





Transaction States in DBMS

1. **Active:** This is the initial state of every transaction. In this state, the transaction is being executed. The transaction remains in this state as long as it is executing SQL statements.
2. **Partially Committed:** When a transaction executes its final statement, it is said to be in a 'partially committed' state. At this point, the transaction has passed the modification phase but has not yet been committed to the database. If a failure occurs at this stage, the transaction will roll back.
3. **Failed:** If a transaction is in a 'partially committed' state and a problem occurs that prevents the transaction from committing, it is said to be in a 'failed' state. When a transaction is in a 'failed' state, it will trigger a rollback operation.
4. **Aborted:** If a transaction is rolled back and the database is restored to its state before the transaction began, the transaction is said to be 'aborted.' After a transaction is aborted, it can be restarted again, but this depends on the policy of the transaction management component of the DBMS.
5. **Committed:** When a transaction is in a 'partially committed' state and the commit operation is successful, it is said to be 'committed.' At this point, the transaction has completed its execution and all of its updates are permanently stored in the database.
6. **Terminated:** After a transaction reaches the 'committed' or 'aborted' state, it is said to be 'terminated.' This is the final state of a transaction.

Note: The actual terms and the number of states may vary depending on the specifics of the DBMS and the transaction processing model it uses. However, the fundamental principles remain the same.

Transaction States

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

Property of Transaction:

- Atomicity
- Consistency
- Isolation
- Durability

Atomicity

- States that all operations of the transaction take place at once if not, the transactions are 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.

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 anew 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.

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 T1ends.
- 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.
-

Implementing of Atomicity and Durability

The recovery-management component of a database system can support atomicity and durability by a variety of schemes.

E.g. the **shadow-database scheme**:

Shadow copy

- In the shadow-copy scheme, a transaction that wants to update the database first creates a complete copy of the database.
- All updates are done on the new database copy, leaving the original copy, the shadow copy, untouched. If at any point the transaction has to be aborted, the system merely deletes the new copy. The old copy of the database has not been affected.
- This scheme is based on making copies of the database, called shadow copies, assumes that only one transaction is active at a time.
- The scheme also assumes that the database is simply a file on disk. A pointer called db pointer is maintained on disk; it points to the current copy of the database.

Implementation of Atomicity and Durability in DBMS

Implementation Atomicity

- When a transaction starts, changes are not immediately written to the original data pages. Instead, new copies of the data pages are created and modified. The original data remains untouched.
- If the transaction fails or needs to be aborted for some reason, the DBMS simply discards the new pages. Since the original data hasn't been altered, atomicity is maintained. No changes from the failed transaction are visible.

Implementation Durability

- Once a transaction is completed successfully, the directory that was pointing to the original pages is updated to point to the modified pages. This switch can be done atomically, ensuring durability.
- After the switch, the new pages effectively become the current data, while the old pages can be discarded or archived.
- Because the actual switching of the pointers is a very fast operation, committing a transaction can be done quickly, ensuring that once a transaction is declared complete, its changes are durable.

Shadow Database scheme

The Shadow Database scheme is a technique used for ensuring atomicity and durability in a database system, two of the ACID properties. The core idea is to keep a shadow copy of the database and make changes to a separate copy. Once the transaction is complete and it's time to commit, the system switches to the new copy, effectively making it the current database. This allows for easy recovery and ensures data integrity even in the case of system failures.

Working Principle of Shadow Database

- **Initial State:** Initially, the database is in a consistent state. Let's call this the "Shadow" database.

- **Transaction Execution:** When transactions are executed, they are not applied directly to the shadow database. Instead, they are applied to a separate copy of the database.
- **Commit:** After a transaction is completed successfully, the system makes the separate copy the new shadow database.
- **Rollback:** If a transaction cannot be completed successfully, the system discards the separate copy, effectively rolling back all changes.
- **System Failure:** If a system failure occurs in the middle of a transaction, the original shadow database remains untouched and in a consistent state.

Shadow Database scheme Example

Let's consider a simple banking database that has one table named **`Account`** with two fields: **`AccountID`** and **`Balance`**.

Shadow Database - Version 1

```
| AccountID | Balance |
|-----------|---------|
| 1         | 1000    |
| 2         | 2000    |
```

1. Transaction Start: A transaction starts to transfer 200 from AccountID 1 to AccountID 2.

2. Separate Copy: A separate copy of the database is made and the changes are applied to it.

Modified Copy

```
| AccountID | Balance |
|-----------|---------|
| 1         | 800     |
| 2         | 2200    |
```

3. Commit: The transaction completes successfully. The modified copy becomes the new shadow database.

Shadow Database - Version 2

```
| AccountID | Balance |
|-----------|---------|
| 1         | 800     |
| 2         | 2200    |
```

4. System Failure: If the system crashes after this point, the shadow database (Version 2) is already in a consistent state, ensuring durability and atomicity.

Shadow Database scheme Pros

- **Atomicity:** All changes are either committed entirely or not at all.
- **Durability:** Once changes are committed, they are permanent, even in the case of system failures.

Shadow Database scheme Cons

- **Disk I/O:** Requires more disk operations because of the separate copy.
- **Concurrency:** More complex to implement in a multi-user environment.

Shadow databases are particularly useful for systems where atomicity and durability are more critical than performance, such as in financial or medical databases.

Concurrent Executions in DBMS

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 in DBMS

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 said to be non serial which result in inconsistency or conflicts in the data.

What is serializability? How it is tested?

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.

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 in DBMS

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)1() 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 Tj reads the result of a write by transaction Ti in one schedule, then Tj should read the result of a write by Ti 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).

Let's understand view serializability with an example:

Consider two transactions T1 and T2:
Schedule 1(S1):

```
| Transaction T1      | Transaction T2      |
|---------------------|---------------------|
| Write(A)            |                     |
|                     | Read(A)             |
|                     | Write(B)            |
```

```
| Read(B)              |                      |
| Write(B)             |                      |
| Commit               | Commit               |
```

Schedule 2(S2):

```
| Transaction T1       | Transaction 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 in DBMS

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.

Importance of 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 Ti�? and Tj??, if Tj?? reads a data item previously written by Ti??, then Ti?? must commit before Tj?? 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 T1?1 and T2?2.

```
| Transaction T1      | Transaction T_2     |
|---------------------|---------------------|
| Write(A)            |                     |
|                     | Read(A)             |
| Commit              |                     |
|                     | Write(B)            |
|                     | Commit              |
```

In the above schedule, T2?2 reads a value written by T1?1, but T1?1 commits before T2?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.

```
| Transaction T1      | Transaction 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:

```
| Transaction T1      | Transaction T2      |
|---------------------|---------------------|
| Write(A)            |                     |
|                     | Read(A)             |
|                     | Write(B)            |
```

```
| Abort(some failure) |                     |
| Rollback            |                     |
|                     | Rollback (because it read uncommitted changes from T1)
```

Here, $T_2$2 reads an uncommitted value of A written by $T_1$1. When $T_1$1 fails and is rolled back, $T_2$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$1 and $T_2$2:

```
| Transaction T1      | Transaction T2      |
|---------------------|---------------------|
| Write(A)            |                     |
| Commit              |                     |
|                     | Read(A)             |
|                     | Write(B)            |
|                     | Commit              |
```

In this schedule, $T_2$2 reads the value of A only after $T_1$1 has committed. Thus, even if $T_1$1 were to fail before committing (not shown in this schedule), it would not affect $T_2$2. This means there's no risk of cascading rollback in this schedule.

Implementation of Isolation in DBMS

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	Unrepetable Read
Serializable	NO	NO
Repeatable Read	NO	NO
Read Committed	NO	Maybe
Read Uncommitted	Maybe	Maybe

Implementation of Isolation

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.

Lock Based Protocols in DBMS

Why Do We Need Locks?

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.

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)
           |  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 in DBMS

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$\diamond$1 tries to write item A$\diamond$.

According to the write rule, T1$\diamond$1 can't write item A$\diamond$ after T2$\diamond$2 has written it, because TS(T2)>TS(T1)$\diamond\diamond$($\diamond$2)>$\diamond\diamond$($\diamond$1). Thus, T1$\diamond$1's write operation will be rejected.

Example-2

Suppose two transactions T1$\diamond$1 and T2$\diamond$2 with timestamps 5 and 10 respectively:

1. T2$\diamond$2 writes item A$\diamond$.
2. T1$\diamond$1 tries to read item A$\diamond$.

According to the read rule, T1$\diamond$1 can't read item A$\diamond$ after T2$\diamond$2 has written it, as TS(T2)>TS(T1)$\diamond\diamond$($\diamond$2)>$\diamond\diamond$($\diamond$1). So, T1$\diamond$1'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 $T1_1$ and $T2_2$:

- Both transactions read an item $A_$ with a value of 10.
- $T1_1$ updates its local copy of $A_$ to 12.
- $T2_2$ updates its local copy of $A_$ to 15.
- $T1_1$ reaches the validation phase and is validated successfully (because there's no other transaction that has written to $A_$ since $T1_1$ read it).
- $T1_1$ moves to the write phase and updates $A_$ in the database to 12.

- $T2_2$ reaches the validation phase. The system realizes that since $T2_2$ read item $A_$, another transaction (i.e., $T1_1$) has written to $A_$. Therefore, $T2_2$ fails validation.
- $T2_2$ 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.

(OR)

Validation Based Protocol

Validation phase is also known as optimistic concurrency control technique. In the validation based protocol, the transaction is executed in the following three phases:

1. **Read phase:** In this phase, the transaction T is read and executed. It is used to read the value of various data items and stores them in temporary local variables. It can perform all the write operations on temporary variables without an update to the actual database.
2. **Validation phase:** In this phase, the temporary variable value will be validated against the actual data to see if it violates the serializability.

3. **Write phase:** If the validation of the transaction is validated, then the temporary results are written to the database or system otherwise the transaction is rolled back.

Here each phase has the following different timestamps:

Start(Ti): It contains the time when Ti started its execution.

Validation (T_i): It contains the time when Ti finishes its read phase and starts its validation phase.

Finish(Ti): It contains the time when Ti finishes its write phase.

- This protocol is used to determine the time stamp for the transaction for serialization using the time stamp of the validation phase, as it is the actual phase which determines if the transaction will commit or rollback.
- Hence TS(T) = validation(T).
- The serializability is determined during the validation process. It can't be decided in advance.
- While executing the transaction, it ensures a greater degree of concurrency and also less number of conflicts.
- Thus it contains transactions which have less number of rollbacks.

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)

The Scheme operates as follows:-

a. A Transaction must first lock the Root Node and it can be locked in any mode.

b. Locks are granted as per the Compatibility Matrix indicated above.

c. A Transaction can lock a node in S or IS mode if it has already locked all the predecessor nodes in IS or IX mode.

d. A Transaction can lock a node in X or IX or SIX mode if it has already locked all the predecessor nodes in SIX or IX mode.

e. A transaction must follow two-phase locking. It can lock a node, only if it has not previously unlocked a node. Thus, schedules will always be conflict-serializable.

f. Before it unlocks a node, a Transaction has to first unlock all the children nodes of that node. Thus, locking will proceed in top-down manner and unlocking will proceed in bottom-up manner. This will ensure the resulting schedules to be deadlock-free.

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.

(OR)

Multiple Granularity

Let's start by understanding the meaning of granularity.

Granularity: It is the size of data item allowed to lock.

Multiple Granularity:

- It can be defined as hierarchically breaking up the database into blocks which can be locked.
- The Multiple Granularity protocol enhances concurrency and reduces lock overhead.
- It maintains the track of what to lock and how to lock.
- It makes easy to decide either to lock a data item or to unlock a data item. This type of hierarchy can be graphically represented as a tree.

For example: Consider a tree which has four levels of nodes.

- The first level or higher level shows the entire database.
- The second level represents a node of type area. The higher level database consists of exactly these areas.
- The area consists of children nodes which are known as files. No file can be present in more than one area.
- Finally, each file contains child nodes known as records. The file has exactly those records that are its child nodes. No records represent in more than one file.
- Hence, the levels of the tree starting from the top level are as follows:
    1. Database
    2. Area
    3. File
    4. Record

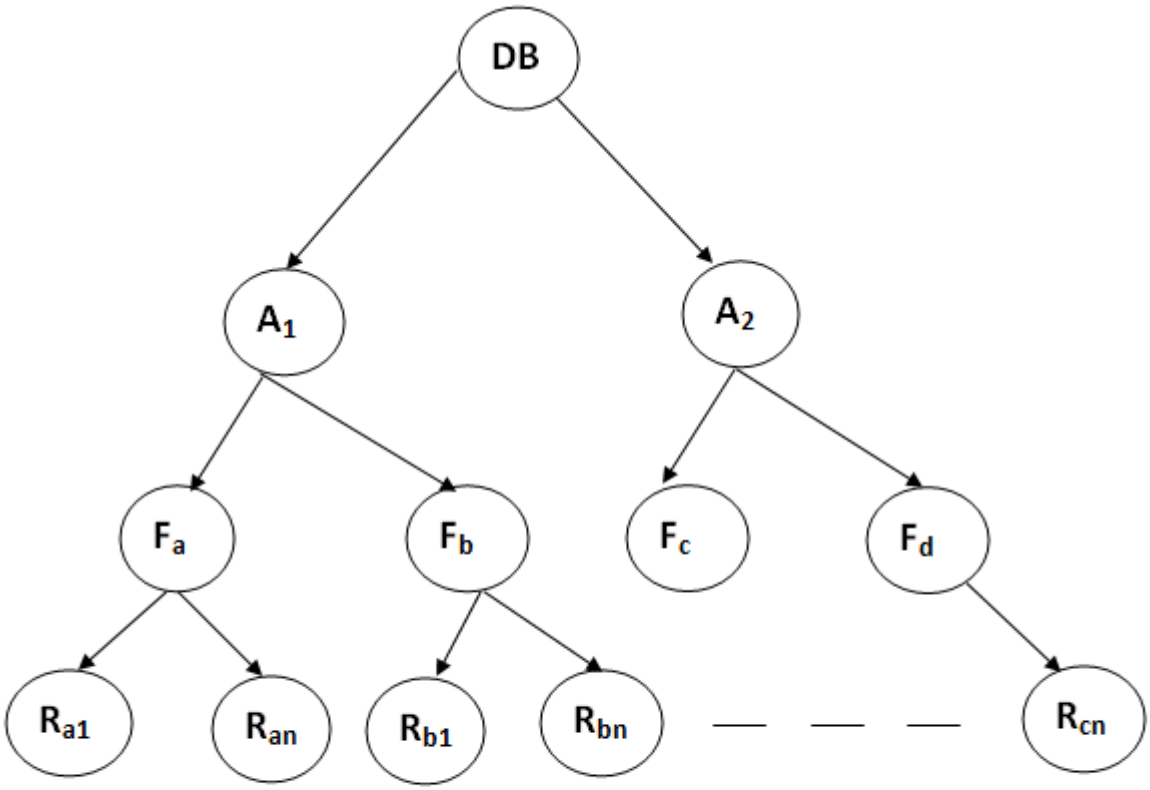


Figure: Multi Granularity tree Hierarchy

In this example, the highest level shows the entire database. The levels below are file, record, and fields.

There are three additional lock modes with multiple granularity:

Intention Mode Lock

Intention-shared (IS): It contains explicit locking at a lower level of the tree but only with shared locks.

Intention-Exclusive (IX): It contains explicit locking at a lower level with exclusive or shared locks.

Shared & Intention-Exclusive (SIX): In this lock, the node is locked in shared mode, and some node is locked in exclusive mode by the same transaction.

Compatibility Matrix with Intention Lock Modes: The below table describes the compatibility matrix for these lock modes:

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	X
IX	✓	✓	X	X	X
S	✓	X	✓	X	X
SIX	✓	X	X	X	X
X	X	X	X	X	X

It uses the intention lock modes to ensure serializability. It requires that if a transaction attempts to lock a node, then that node must follow these protocols:

- Transaction T1 should follow the lock-compatibility matrix.
- Transaction T1 firstly locks the root of the tree. It can lock it in any mode.
- If T1 currently has the parent of the node locked in either IX or IS mode, then the transaction T1 will lock a node in S or IS mode only.
- If T1 currently has the parent of the node locked in either IX or SIX modes, then the transaction T1 will lock a node in X, SIX, or IX mode only.
- If T1 has not previously unlocked any node only, then the Transaction T1 can lock a node.
- If T1 currently has none of the children of the node-locked only, then Transaction T1 will unlock a node.

Observe that in multiple-granularity, the locks are acquired in top-down order, and locks must be released in bottom-up order.

- If transaction T1 reads record R_{a9} in file F_a, then transaction T1 needs to lock the database, area A_1 and file F_a in IX mode. Finally, it needs to lock R_{a2} in S mode.
- If transaction T2 modifies record R_{a9} in file F_a, then it can do so after locking the database, area A_1 and file F_a in IX mode. Finally, it needs to lock the R_{a9} in X mode.
- If transaction T3 reads all the records in file F_a, then transaction T3 needs to lock the database, and area A in IS mode. At last, it needs to lock F_a in S mode.
- If transaction T4 reads the entire database, then T4 needs to lock the database in S mode.

Recovery and Atomicity in dbms

- When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items.
- But according to ACID properties of DBMS, atomicity of transactions as a whole must be maintained, that is, either all the operations are executed or none.
- Database recovery means recovering the data when it get deleted, hacked or damaged accidentally.
- Atomicity is must whether is transaction is over or not it should reflect in the database permanently or it should not effect the database at all.

When a DBMS recovers from a crash, it should maintain the following −

- It should check the states of all the transactions, which were being executed.
- A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.
- It should check whether the transaction can be completed now or it needs to be rolled back.
- No transactions would be allowed to leave the DBMS in an inconsistent state.

There are two types of techniques, which can help a DBMS in recovering as well as maintaining the atomicity of a transaction −

- Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.
- Maintaining shadow paging, where the changes are done on a volatile memory, and later, the actual database is updated

Log-Based Recovery

Log-based recovery is a widely used approach in database management systems to recover from system failures and maintain atomicity and durability of transactions. The fundamental idea behind log-based recovery is to keep a log of all changes made to the database, so that after a failure, the system can use the log to restore the database to a consistent state.

How Log-Based Recovery Works

1. Transaction Logging:

For every transaction that modifies the database, an entry is made in the log. This entry typically includes:

- **Transaction ID**: A unique identifier for the transaction.
- **Data item identifier**: Identifier for the specific item being modified.
- **OLD value**: The value of the data item before the modification.
- **NEW value**: The value of the data item after the modification.

We represent an update log record as <Ti�� , Xj�� , V1�1, V2�2>, indicating that transaction Ti�� has performed a write on data item Xj��. Xj�� had value V1�1 before the write, and has value V2�2 after the write. Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction. Among the types of log records are:

- <Ti�� start>. Transaction Ti has started.
- <Ti�� commit>. Transaction Ti has committed.
- <Ti�� abort>. Transaction Ti has aborted.

2. Writing to the Log

Before any change is written to the actual database (on disk), the corresponding log entry is stored. This is called the **Write-Ahead Logging (WAL)** principle. By ensuring that the log is written first, the system can later recover and apply or undo any changes.

3. Checkpointing

Periodically, the DBMS might decide to take a checkpoint. A checkpoint is a point of synchronization between the database and its log. At the time of a checkpoint:

- All the changes in main memory (buffer) up to that point are written to disk.
- A special entry is made in the log indicating a checkpoint. This helps in reducing the amount of log that needs to be scanned during recovery.

4. Recovery Process

- Redo: If a transaction is identified (from the log) as having committed but its changes have not been reflected in the database (due to a crash before the

changes could be written to disk), then the changes are reapplied using the 'After Image' from the log.

- Undo: If a transaction is identified as not having committed at the time of the crash, any changes it made are reversed using the 'Before Image' in the log to ensure atomicity.

5. Commit/Rollback

Once a transaction is fully complete, a commit record is written to the log. If a transaction is aborted, a rollback record is written, and using the log, the system undoes any changes made by this transaction.

Benefits of Log-Based Recovery

- **Atomicity:** Guarantees that even if a system fails in the middle of a transaction, the transaction can be rolled back using the log.
- **Durability:** Ensures that once a transaction is committed, its effects are permanent and can be reconstructed even after a system failure.
- **Efficiency:** Since logging typically involves sequential writes, it is generally faster than random access writes to a database.

Shadow paging - Its Working principle

Shadow Paging is an alternative disk recovery technique to the more common logging mechanisms. It's particularly suitable for database systems. The fundamental concept behind shadow paging is to maintain two page tables during the lifetime of a transaction: the current page table and the shadow page table.

Here's a step-by-step breakdown of the working principle of shadow paging:

Initialization

When the transaction begins, the database system creates a copy of the current page table. This copy is called the shadow page table.

The actual data pages on disk are not duplicated; only the page table entries are. This means both the current and shadow page tables point to the same data pages initially.

During Transaction Execution

When a transaction modifies a page for the first time, a copy of the page is made. The current page table is updated to point to this new page.

Importantly, the shadow page table remains unaltered and continues pointing to the original, unmodified page.

Any subsequent changes by the same transaction are made to the copied page, and the current page table continues to point to this copied page.

On Transaction Commit

Once the transaction reaches a commit point, the shadow page table is discarded, and the current page table becomes the new "truth" for the database state.

The old data pages that were modified during the transaction (and which the shadow page table pointed to) can be reclaimed.

Recovery after a Crash

If a crash occurs before the transaction commits, recovery is straightforward. Since the original data pages (those referenced by the shadow page table) were never modified, they still represent a consistent database state.

The system simply discards the changes made during the transaction (i.e., discards the current page table) and reverts to the shadow page table.

Recovery with Concurrent Transactions in DBMS

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.

Starvation in DBMS

Starvation is a situation when one **transaction** keeps on waiting for another transaction to release the lock. This is also called **LiveLock**. As we already learned in transaction management that a transaction acquires lock before performing a write operation on data item, if the data item is already locked by another transaction then the transaction waits for the lock to be released. **In starvation situation a transaction waits for another transaction for an infinite period of time.**

Why Starvation occurs?

1. If the transactions are **not having a priority set**. Generally the older transaction is given higher priority so that the transaction waiting for a longer period of time gets the lock sooner than the transaction waiting for a short period of time. If the priorities are not set then a transaction can keep on waiting while other transactions are continuously acquiring the lock on data item.

2. **Resource leak:** When a transaction does not release the lock after it has acquired the lock on a particular data item.

3. **Denial of service attack:** A Denial-of-Service (DoS) attack is an attack that is meant to shut down a machine or network, making it inaccessible to the

users. DoS attack make the data item engaged so that the transaction are not able to acquire the locks.

Starvation Example

Let's say there are three transaction T1, T2 and T3 waiting to acquire lock on a data item 'X'. System grants a lock to the transaction T1, the other two transaction T2 and T3 are waiting for the lock to be released.

Once the transaction T1 release the lock, the lock is granted to transaction T3, now transaction T2 is waiting for the lock to be released.

While transaction T3 is performing an operation on 'X', a new transaction T4 enters into the system and wait for the lock. The system grants the lock to T4. This way new transactions keep on entering into the system and acquiring the lock on 'X' while the older transaction T2 keeps on waiting.

How to solve the starvation problem in DBMS?

1. **Increase priority:** One way of fixing the starvation issue is to grant higher priority to the older transaction. This way the transaction that requested for the lock first will have higher priority than the transaction that requested for the lock later.

The drawback to this solution is that a faulty transaction keeps on acquiring the lock and failing so it never gets completed and remains there with the higher priority than other transactions, thus keeps on getting the lock on a particular data item.

2. **By changing the victim selection algorithm:** In the above solution, we saw a drawback where a victim transaction keeps on getting the lock. By lowering the priority of a victim transaction, we can fix the drawback of above solution.

3. **FCFS (First come first serve):** In this approach, the transaction that entered into the system first, gets the lock first. This way no transaction keeps on waiting.

4. **Wait-die Scheme:** If a transaction requests a lock on data item that is acquired by another transaction then system checks for the timestamp and allow the older transaction to wait for the data item.

5. **Wound-wait Scheme:** In this scheme, if older transaction requests for the lock which is held by younger transaction then the system kills the younger transaction and grants the lock to older transaction.

The **killed younger transaction is restarted with a specific delay but with same timestamp**, this make sure that after some time when this transaction is old enough it can acquire the lock on particular data item.

These both schemes can be represented in a tabular format like this:

SITUATION	**WAIT – DIE**	**WOUND- WAIT**
Older process needs a resource held by younger process	**Older process** waits	**Younger process** dies
Younger process needs a resource held by older process	**Younger process** dies	**Younger process** waits