**ACID properties**

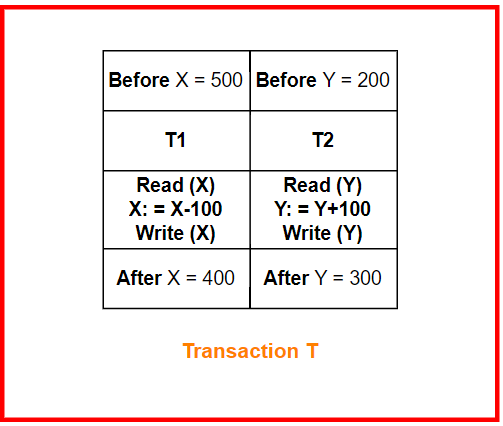
A transaction in a Database Management System (DBMS) is a single logical unit of work that accesses and possibly alters the contents of a database. Transactions use read-and-write operations to manage data. Certain properties are adhered to in order to ensure database consistency before and after a transaction. These properties are known as ACID properties.

**Atomicity**

Atomicity means that either the entire transaction is executed, or none of it is. There is no partial execution. Each transaction is seen as a single unit that either completes fully or does not execute at all. This involves two operations:

* **Abort:** If a transaction aborts, any changes made to the database are not visible.
* **Commit:** If a transaction is committed, the changes made are visible.

Atomicity is also known as the "All or nothing rule."



**Consistency**

Consistency means that integrity constraints must be maintained so that the database remains consistent before and after the transaction. It ensures the correctness of the database.

Example: Referring to the previous example, the total amount before and after the transaction must be the same. If initially, Account X has £500 and Account Y has £200, then:

* Total before transaction T = £500 + £200 = £700
* Total after transaction T = £400 + £300 = £700

The database remains consistent. If T1 completes but T2 fails, the transaction T is incomplete, causing inconsistency.

**Isolation**

Isolation ensures that multiple transactions can occur concurrently without causing database inconsistency. Transactions operate independently without interference. Changes made in a particular transaction are not visible to other transactions until those changes are written to memory or committed.

Example: Let X = £500, Y = £500. Consider two transactions T and T'. If T executes until Read(Y) and then T' starts, operations may interleave, causing T' to read the correct value of X but an incorrect value of Y. Thus, the computed sum by T' (X + Y = 500 + 500 = 1000) will be inconsistent with the sum after T completes (X + Y = 500 + 450 = 950). This inconsistency highlights the need for isolation, ensuring changes are only visible after being committed.

**Durability**

Durability guarantees that once a transaction has been completed, its updates and modifications are stored on disk and persist even in the case of a system failure. These updates become permanent and are stored in non-volatile memory.

Example: Once a transfer of £100 from Account X to Account Y is committed, the changes are permanently recorded. If the system crashes immediately after the transaction, the changes will not be lost and will be reflected when the system is restored.

**Advantages of ACID Properties in DBMS:**

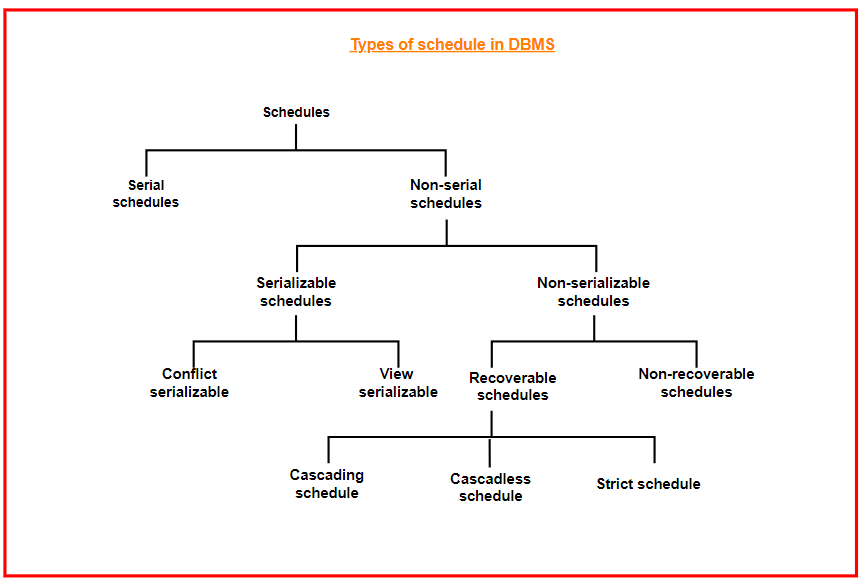
* Data Consistency: Maintains consistent and accurate data after transaction execution.
* Data Integrity: Ensures that changes to the database are permanent.
* Concurrency Control: Manages multiple transactions occurring simultaneously, preventing interference.
* Recovery: Enables the system to recover data up to the point of failure.

**Disadvantages of ACID Properties in DBMS:**

* Performance: This can cause performance overhead due to additional processing.
* Scalability: This may pose scalability challenges in large distributed systems.
* Complexity: Implementation can increase system complexity and require significant resources

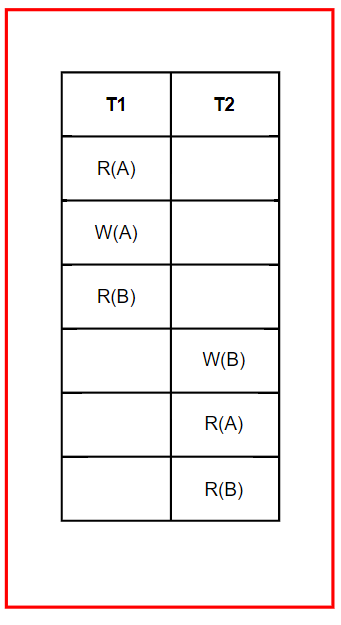
**Schedule(SERIAL, PARALLEL)**

In database management systems (DBMS), scheduling is the process of organizing and executing transactions in a specific sequence to ensure that the operations do not interfere with each other, maintaining the consistency and integrity of the database.



**Serial Schedules**

Serial schedules execute transactions one after another without interleaving. This means no transaction starts until the currently running transaction has finished. This approach ensures no conflicts but does not take advantage of the potential concurrency benefits.



Here, T1 completes all its operations before T2 starts. This is a serial schedule since T1 and T2 do not overlap.

**Non-serial schedules**

A non-serial schedule interleaves the operations of multiple transactions. This can lead to concurrency issues but aims to keep the result consistent with that of a serial schedule. Unlike serial schedules, where one transaction waits for another to complete, in non-serial schedules, transactions proceed concurrently. Non-serial schedules can be either serialisable or non-serializable.

**2.1. Serialisable Schedules**

Serialisable schedules ensure database consistency by verifying that the interleaved execution is equivalent to serial execution. This is crucial in non-serial scheduling, whereas serial schedules inherently maintain consistency. Serialisable schedules enhance resource utilization and CPU throughput. There are two types:

**2.1.1. Conflict Serialisable**

A schedule is conflict serialisable if it can be transformed into a serial schedule by swapping non-conflicting operations. Operations are conflicting if they:

* Belong to different transactions.
* Operate on the same data item.
* At least one operation is a write.

**2.1.2. View Serialisable**

A schedule is view serialisable if it is view equivalent to a serial schedule, meaning it produces the same outcome as a serial schedule. While all conflict serialisable schedules are view serialisable, the reverse is not always true, especially if blind writes are involved.

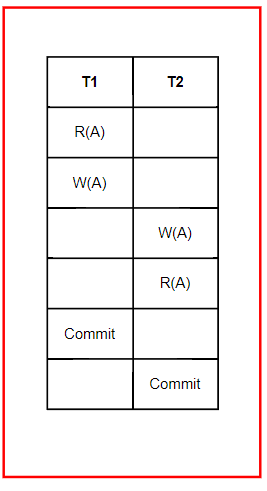
* **Example of Blind Write:**
  + T1 writes X without reading it first.
  + T2 also writes X without reading it.
  + This can be view serializable but **not** conflict serializable.

**2.2. Non-serialisable schedule**

Non-serialisable schedules can be divided into two types: Recoverable and Non-recoverable Schedules.

**2.2.1. Recoverable schedule**

A recoverable schedule ensures that a transaction commits only after all transactions whose changes it has read also commit. In other words, if transaction Tj reads a value updated by transaction Ti, then Tj must commit only after Ti commits.



This is a recoverable schedule since T1 commits before T2, making the value read by T2 correct.

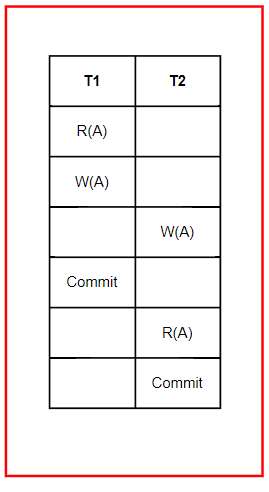
**2.2.1.1. Cascading Schedule**

Also known as Avoids Cascading Aborts/Rollbacks (ACA). When a failure in one transaction leads to the rollback or abort of other dependent transactions, it is referred to as a cascading rollback or cascading abort.

If a transaction **aborts**, other transactions that read its uncommitted data **must also abort**.

**2.2.1.2. Cascadeless Schedule**

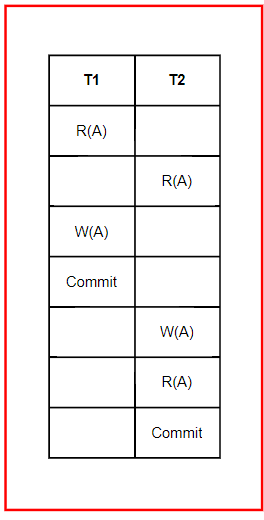
In cascadeless schedules, transactions read values only after all transactions whose changes they are going to read commit. This avoids a series of transaction rollbacks due to a single transaction abort. To prevent cascading aborts, a transaction is disallowed from reading uncommitted changes from another transaction in the same schedule.



This schedule is cascadeless since T2 reads the updated value of A only after T1 commits.

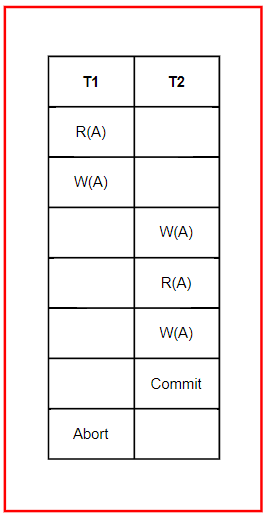
**2.2.1.3. Strict Schedule**

A schedule is strict if for any two transactions Ti and Tj, if a write operation of Ti precedes a conflicting operation of Tj (either read or write), then the commit or abort event of Ti also precedes that conflicting operation of Tj. In other words, Tj can read or write a value updated by Ti only after Ti commits or aborts.



**2.2.2. Non-Recoverable schedule**

A non-recoverable schedule allows a transaction to commit based on an uncommitted change, which can lead to inconsistency if the initial transaction later aborts.



Here, T2 reads the value of A written by T1 and commits. If T1 later aborts, the value read by T2 is incorrect, but since T2 has committed, this schedule is non-recoverable

**Isolation levels and its types**

In database management systems (DBMS), isolation levels are important for keeping data consistent while allowing multiple transactions to happen at the same time. Isolation is part of the ACID properties: Atomicity, Consistency, Isolation, and Durability. It ensures that transactions do not interfere with each other.

**Phenomena Related to Isolation**

* **Dirty Read**: This happens when a transaction reads data that hasn’t been saved yet. For example, if Transaction T1 updates a row but hasn't committed it, and Transaction T2 reads that row, T2 gets uncommitted data. If T1 rolls back its changes, the data T2 read becomes invalid.
* **Non-Repeatable Read**: This occurs when a transaction reads the same row twice and gets different values. If Transaction T1 reads data, and then Transaction T2 updates that data and commits it, T1 will see different data if it reads the row again.
* **Phantom Read**: This happens when two identical queries are run but return different results. For example, if Transaction T1 retrieves rows based on a condition, and then Transaction T2 adds new rows that meet this condition, T1 will get different results when it runs the query again.

**SQL Isolation Levels**

The SQL standard defines four isolation levels that describe how changes made by one transaction are visible to others:

* **Read Uncommitted**: The lowest level, where transactions can read uncommitted changes from others. This can lead to dirty reads, non-repeatable reads, and phantom reads.
* **Read Committed**: This level ensures that any data read is committed at the time it is read, preventing dirty reads. However, it can still result in non-repeatable reads and phantom reads.
* **Repeatable Read**: Guarantees that if a transaction reads a row, it will see the same data throughout its duration, preventing dirty reads and non-repeatable reads, but still allowing phantom reads.
* **Serializable**: The highest isolation level, where transactions are executed one after another, ensuring no dirty reads, non-repeatable reads, or phantom reads.

**Choosing the Appropriate Isolation Level**

When deciding on an isolation level, consider the following:

* **Higher Isolation Levels**: These offer stronger consistency but may slow down the system due to longer lock times.
* **Lower Isolation Levels**: These improve speed and concurrency but can cause data inconsistencies.

**Additional Isolation Features**

Some DBMSs have extra isolation features like:

* **Snapshot Isolation**: Allows transactions to work with a snapshot of the database at a specific time.
* **Multi-Version Concurrency Control (MVCC)**: Maintains multiple versions of data for better concurrency.

**Advantages of Isolation Levels**

* **Improved Concurrency**: Multiple transactions can run at the same time without affecting each other.
* **Data Consistency Control**: Helps manage how consistent the data should be.
* **Reduced Data Anomalies**: Helps prevent dirty reads, non-repeatable reads, and phantom reads.
* **Flexibility**: Different levels offer flexibility for various application needs.

**Disadvantages of Isolation Levels**

* **Increased Overhead**: More checks and locks can slow down the system.
* **Decreased Concurrency**: Higher levels can reduce the number of transactions that can happen simultaneously.
* **Limited Support**: Not all DBMS support all isolation levels, which can affect portability.
* **Complexity**: Different isolation levels add complexity to database applications.

**Examples of Isolation Phenomena**

* **Dirty Read**: If T1 updates a row and doesn’t commit it, T2 reads that row. If T1 rolls back, T2 has read invalid data.
* **Non-Repeatable Read**: T1 reads a row. T2 updates and commits it. T1 reads it again and sees a different value.
* **Phantom Read**: T1 retrieves rows based on a condition. T2 adds rows meeting that condition. T1 runs the query again and gets new results

**Serializability and concurrency control**

In database management, **serializability** and **concurrency control** are essential concepts that ensure the consistency and accuracy of databases when multiple transactions occur simultaneously. These techniques help maintain data integrity and avoid issues like data anomalies.

**What is Serializability?**

**Serializability** is a concept that ensures the results of executing multiple transactions concurrently are the same as if they were executed one after the other, in some order. It helps in achieving a consistent database state, even when transactions overlap in time.

**Types of Serializability**

There are different forms of serializability that help in determining whether a schedule (order of transactions) is serializable:

* **Conflict Serializability:** A schedule is conflict-serializable if it can be converted into a serial schedule by swapping non-conflicting operations.
* **View Serializability:** A schedule is view-serializable if it preserves the same view as a serial schedule, meaning that transactions see the same data, even if the order of operations differs.

**Example:** Suppose two transactions, T1 and T2, read and write the same data item. If T1 writes before T2 reads, conflict serializability ensures that T2 will get the value written by T1, as if they executed in that order.

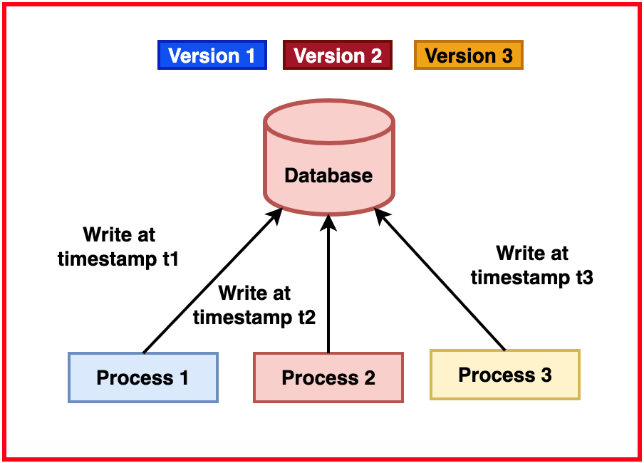
**Challenges of Serializability**

Ensuring serializability is critical but can be challenging due to:

* **Performance Overheads:** Serializability can slow down the system as transactions wait for others to finish.
* **Complexity:** Determining whether a schedule is serializable can be complex, especially for large numbers of transactions.

**What is Concurrency Control?**

**Concurrency Control** refers to the methods used to manage simultaneous operations on a database without causing conflicts. It ensures that database transactions are executed in a way that maintains consistency and isolation.



The primary goal of concurrency control is to prevent issues such as:

* **Lost Update:** Occurs when two transactions update the same data simultaneously, resulting in the loss of one of the updates.
* **Dirty Read:** When a transaction reads uncommitted data from another transaction, which might later be rolled back.
* **Non-Repeatable Read:** Occurs when a transaction reads the same data multiple times but gets different values each time due to updates by other transactions.

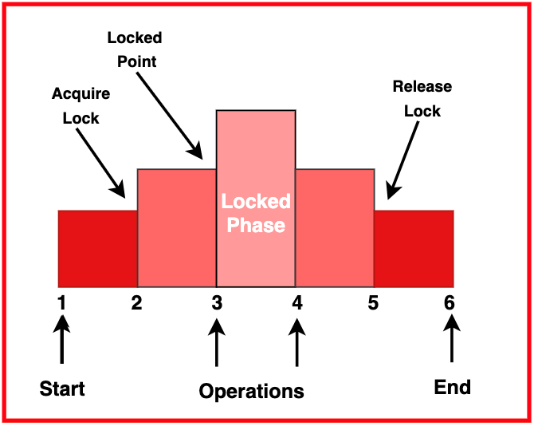
**Techniques for Concurrency Control**

There are various techniques to implement concurrency control in databases:

* **1. Lock-Based Protocols**

**Lock-Based Protocols** use locks to control access to data. A transaction must acquire a lock before it can read or write a data item.

Locks can be **shared** (allowing read-only access) or **exclusive** (allowing write access). Two-phase locking (2PL) is a common lock-based protocol.



**Example:** In an e-commerce database, if one transaction locks a product record for updating, another transaction cannot access it until the lock is released, ensuring consistency.

* **2. Timestamp-Based Protocols**

**Timestamp-Based Protocols** assign a unique timestamp to each transaction and use these timestamps to determine the order of execution. Transactions are executed based on their timestamps, maintaining serializability.

**Example:** A banking system uses timestamps to ensure that older transactions are processed before newer ones, preventing conflicts between transactions that update the same account.

* **3. Optimistic Concurrency Control**

**Optimistic Concurrency Control** assumes that transactions do not conflict and checks for conflicts only at the time of commit. If a conflict is detected, the transaction is rolled back.

**Example:** A social media application allows users to update their profiles simultaneously. Conflicts are checked when the data is saved, and users are prompted to try again if a conflict occurs.

* **4. Multiversion Concurrency Control (MVCC)**

**MVCC** maintains multiple versions of a data item, allowing transactions to read different versions based on their timestamps. It provides high performance and reduces the need for locks.

**Example:** A database for an online streaming platform uses MVCC to allow viewers to read data while updates are being made, ensuring smooth user experience without blocking reads.

**Importance of Serializability and Concurrency Control**

Maintaining serializability and using effective concurrency control techniques help in:

* **Ensuring Data Integrity:** Prevents data corruption by managing concurrent transactions properly.
* **Maximizing System Throughput:** Allows multiple transactions to execute concurrently, increasing the overall performance of the database.
* **Improving User Experience:** Reduces waiting times for users accessing the database, especially in high-traffic applications.

**Applications of Serializability and Concurrency Control**

* **Banking Systems:** Ensures that transactions like deposits and withdrawals are processed accurately, even when multiple users access accounts simultaneously.
* **E-commerce Platforms:** Prevents issues like double booking or inventory mismatches when multiple customers purchase the same item at the same time.
* **Social Media:** Maintains consistency when users interact with posts and comments concurrently, ensuring that data is updated correctly.
* **Online Reservations:** Ensures that booking systems maintain accurate availability data when multiple users attempt to make reservations.

**Locking protocols(shared locks, exclusive locks)**

When multiple transactions occur at the same time, they might try to access the same data. Locking protocols help control how these transactions access data, ensuring that the database stays consistent and correct.

Locking protocols in a Database Management System (DBMS) manage access to data items to maintain consistency and isolation in a multi-user environment. They control how and when locks are applied to avoid conflicts such as lost updates, dirty reads, and uncommitted data.

**Types of Locks**

**1. Shared Locks (S-Locks)**

A shared lock allows multiple transactions to read a data item at the same time but prevents any from writing to it. This lock is also called a read-only lock and is requested using the lock-S instruction.

* **Purpose:** Allows read-only operations to proceed without interference and ensures no modifications occur while data is being read.
* **Characteristics:**
  + Multiple transactions can hold shared locks on the same data item simultaneously.
  + Prevents data modifications while shared locks are held.
* **Example:** Transaction T1 and T2 both want to read the balance of account A. Both can hold a shared lock on account A at the same time.

**2. Exclusive Locks (X-Locks)**

An exclusive lock allows a transaction to both read and write a data item. It ensures that no other transaction can read or write the data item until the exclusive lock is released.

* **Purpose:** Ensures complete control over a data item for modifications and prevents other transactions from accessing the data item.
* **Characteristics:**
  + Only one transaction can hold an exclusive lock on a data item at any time.
  + Blocks both read and write access for other transactions.
* **Example:** Transaction T1 wants to update the balance of account A. It acquires an exclusive lock on account A, preventing any other transaction from accessing account A until T1 releases the lock.

**Concurrency Control Protocols**

Concurrency control protocols allow multiple transactions to happen while ensuring they are conflict/view serializable, recoverable, and sometimes cascadeless. These protocols enforce rules to prevent non-serializable schedules.

**Types of Lock-Based Protocols**

**Simplistic Lock Protocol**

The simplistic lock protocol requires that a transaction must obtain a lock on every data item it accesses before reading or writing. Once the transaction completes all its operations, it releases all the locks.

* **Characteristics:**
  + Easy to implement with a simple rule: acquire a lock before accessing a data item.
  + Uses a single type of lock, limiting concurrent access to data items.
* **Advantages:**
  + Simple to understand and implement.
  + Maintains data consistency by preventing concurrent access to the same data item.
* **Disadvantages:**
  + Limits concurrency and can lead to performance bottlenecks.
  + Does not address deadlocks, where transactions wait indefinitely for each other.

**Pre-Claiming Lock Protocol**

The pre-claiming lock protocol mandates that a transaction must declare and obtain all the locks it will need before any operations are performed.

* **Characteristics:**
  + A transaction must acquire all required locks at the beginning or wait until they are all available.
  + Helps prevent deadlocks by avoiding cyclic dependencies.
* **Advantages:**
  + Prevents deadlocks by acquiring all locks upfront.
  + Straightforward to understand and implement.
* **Disadvantages:**
  + Holding all locks can reduce concurrency, leading to potential performance issues.
  + Locks may be held longer than necessary, leading to inefficient resource use.

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

The Two-Phase Locking protocol divides transaction execution into two phases: the growing phase and the shrinking phase.

* **Growing Phase:** Transactions can acquire locks but cannot release any.
* **Shrinking Phase:** Transactions can release locks but cannot acquire new ones.
* **Characteristics:**
  + Guarantees that transactions are serializable.
  + Can lead to deadlocks, similar to simplistic protocols.
* **Advantages:**
  + Preserves database consistency through serializability.
  + Ensures transactions do not interfere with each other.
* **Disadvantages:**
  + Can lead to deadlocks.
  + Performance overhead due to locking phases.

**Strict Two-Phase Locking (Strict-2PL)**

This is a variant of the 2PL protocol where a transaction must hold all its exclusive locks until it commits or aborts.

* **Characteristics:**
  + Exclusive locks are retained until the transaction completes.
  + Shared locks can be released before the transaction commits.
* **Advantages:**
  + Prevents cascading aborts, ensuring consistency.
  + Simplifies recovery as uncommitted changes are not visible to other transactions.
* **Disadvantages:**
  + Can lead to deadlocks, similar to basic 2PL.
  + Reduced concurrency due to holding exclusive locks.

**Database recovery management**

Database Recovery Management is the process of restoring a database to a correct state after a failure. This ensures that databases can recover from unexpected events like hardware malfunctions, software errors, or human mistakes, maintaining data integrity and minimizing downtime.

**Why is Database Recovery Important?**

**Database recovery** is crucial to ensure that **data integrity** and **availability** are maintained. When a failure occurs, it can cause data loss or corruption. Effective recovery methods help restore data, minimize business interruptions, and ensure that operations can resume smoothly.

**Types of Database Failures**

Understanding the types of failures that can affect a database is key to designing effective recovery strategies. Some common types of failures include:

* **Transaction Failure:** Occurs when a transaction is unable to complete successfully, often due to input errors or logical inconsistencies.
* **System Failure:** Happens when the operating system or database software crashes, causing all active transactions to halt.
* **Media Failure:** Refers to physical damage to storage devices, such as a hard drive failure, that makes stored data inaccessible.
* **Natural Disasters:** Includes events like floods, earthquakes, or fires that can lead to loss of entire database servers.

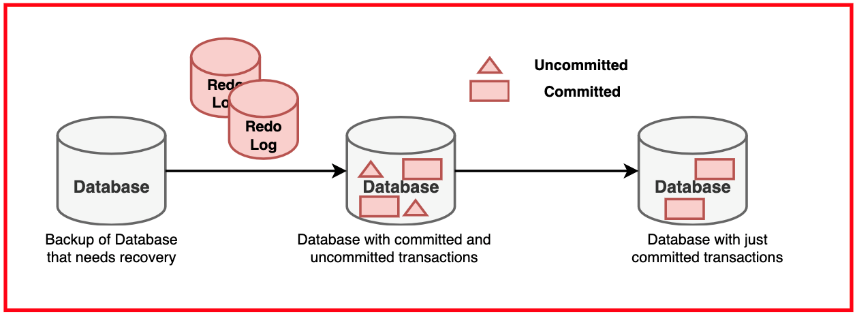
**Techniques for Database Recovery**

There are several techniques to recover a database after a failure. These methods help restore data to a consistent state, ensuring minimal data loss and downtime.

* **1. Backup and Restore**

The **Backup and Restore** method involves creating periodic backups of the database, which can be used to restore data in case of a failure.

**Example:** A company creates daily backups of its customer database. In the event of a system crash, they can use the latest backup to restore the data.



* **2. Log-Based Recovery**

**Log-Based Recovery** keeps a record of all database changes in a log file. If a failure occurs, the log file is used to redo completed transactions and undo incomplete ones.

**Example:** A bank uses log-based recovery to ensure that transactions are either fully completed or rolled back if a failure happens during processing.

* **3. Shadow Paging**

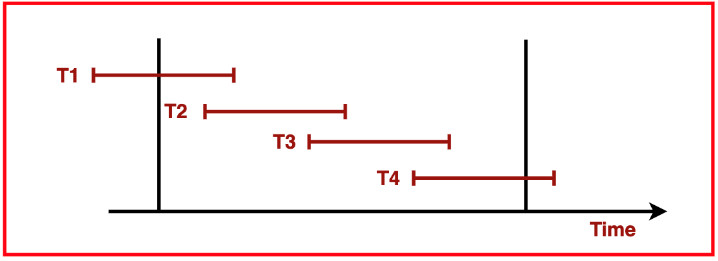
**Shadow Paging** maintains two copies of a database page: a shadow page and a current page. During updates, changes are made to the current page while the shadow page remains intact, providing a recovery point.

**Example:** A manufacturing company uses shadow paging to protect its inventory database. If a failure occurs, the system reverts to the shadow page to restore the previous state.

* **4. Checkpointing**

**Checkpointing** involves creating a snapshot of the database at a particular point in time. During recovery, the system starts from the last checkpoint, reducing the amount of data that needs to be processed.

**Example:** An e-commerce platform creates checkpoints every 10 minutes, so in case of a crash, only transactions after the last checkpoint need to be reprocessed.



* **5. Rollback and Rollforward**

**Rollback** undoes changes made by incomplete transactions, while **Rollforward** applies changes from completed transactions to restore the database to a consistent state.

**Example:** In a banking system, if a money transfer fails midway, rollback reverses the partial transaction. Rollforward can then be used to apply changes from successfully completed transactions.

**Best Practices for Database Recovery**

* **Regular Backups:** Schedule frequent backups to ensure that data can be restored quickly when needed.
* **Store Backups Off-Site:** Keep copies of backups in different locations to protect against natural disasters.
* **Test Recovery Procedures:** Periodically test recovery methods to ensure they are effective and up-to-date.
* **Automate Recovery Processes:** Use automation to speed up recovery and reduce the chance of human error.

**Applications of Database Recovery Management**

* **Financial Services:** Banks and investment firms rely on robust recovery methods to ensure that transactions and records are accurate, even after a system failure.
* **Healthcare:** Hospitals and clinics use recovery management to protect patient data, ensuring that medical records are safe and available during emergencies.
* **Retail:** Retailers use database recovery techniques to safeguard transaction data and inventory records, helping to resume operations quickly after an outage.
* **Government Agencies:** Public sector organizations implement recovery management to maintain the availability of critical data, such as census records and tax information.