**1. Normalization**

A large database defined as a single relation may result in data duplication. This repetition of data may result in:

* Making relations very large.
* It isn't easy to maintain and update data as it would involve searching many records in relation.
* Wastage and poor utilization of disk space and resources.
* The likelihood of errors and inconsistencies increases.

So to handle these problems, we should analyze and decompose the relations with redundant data into smaller, simpler, and well-structured relations that are satisfy desirable properties. Normalization is a process of decomposing the relations into relations with fewer attributes.

## **What is Normalization?**

* Normalization is the process of organizing the data in the database.
* Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate undesirable characteristics like Insertion, Update, and Deletion Anomalies.
* Normalization divides the larger table into smaller and links them using relationships.
* The normal form is used to reduce redundancy from the database table.

Why do we need Normalization?

The main reason for normalizing the relations is removing these anomalies. Failure to eliminate anomalies leads to data redundancy and can cause data integrity and other problems as the database grows. Normalization consists of a series of guidelines that helps to guide you in creating a good database structure.

**Data modification anomalies can be categorized into three types:**

* **Insertion Anomaly:** Insertion Anomaly refers to when one cannot insert a new tuple into a relationship due to lack of data.
* **Deletion Anomaly:** The delete anomaly refers to the situation where the deletion of data results in the unintended loss of some other important data.
* **Updatation Anomaly:** The update anomaly is when an update of a single data value requires multiple rows of data to be updated.

## **Types of Normal Forms:**

Normalization works through a series of stages called Normal forms. The normal forms apply to individual relations. The relation is said to be in particular normal form if it satisfies constraints.

**Following are the various types of Normal forms:**



|  |  |
| --- | --- |
| **Normal Form** | **Description** |
| [1NF](https://www.javatpoint.com/dbms-first-normal-form) | A relation is in 1NF if it contains an atomic value. |
| [2NF](https://www.javatpoint.com/dbms-second-normal-form) | A relation will be in 2NF if it is in 1NF and all non-key attributes are fully functional dependent on the primary key. |
| [3NF](https://www.javatpoint.com/dbms-third-normal-form) | A relation will be in 3NF if it is in 2NF and no transition dependency exists. |
| BCNF | A stronger definition of 3NF is known as Boyce Codd's normal form. |
| [4NF](https://www.javatpoint.com/dbms-forth-normal-form) | A relation will be in 4NF if it is in Boyce Codd's normal form and has no multi-valued dependency. |
| [5NF](https://www.javatpoint.com/dbms-fifth-normal-form) | A relation is in 5NF. If it is in 4NF and does not contain any join dependency, joining should be lossless. |

## **Advantages of Normalization**

* Normalization helps to minimize data redundancy.
* Greater overall database organization.
* Data consistency within the database.
* Much more flexible database design.
* Enforces the concept of relational integrity.

## **Disadvantages of Normalization**

* You cannot start building the database before knowing what the user needs.
* The performance degrades when normalizing the relations to higher normal forms, i.e., 4NF, 5NF.
* It is very time-consuming and difficult to normalize relations of a higher degree.
* Careless decomposition may lead to a bad database design, leading to serious problems.

# 2. First Normal Form (1NF)

* A relation will be 1NF if it contains an atomic value.
* It states that an attribute of a table cannot hold multiple values. It must hold only single-valued attribute.
* First normal form disallows the multi-valued attribute, composite attribute, and their combinations.

**Example:** Relation EMPLOYEE is not in 1NF because of multi-valued attribute EMP\_PHONE.

**EMPLOYEE table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_PHONE** | **EMP\_STATE** |
| 14 | John | 7272826385, 9064738238 | UP |
| 20 | Harry | 8574783832 | Bihar |
| 12 | Sam | 7390372389, 8589830302 | Punjab |

The decomposition of the EMPLOYEE table into 1NF has been shown below:

|  |  |  |  |
| --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_PHONE** | **EMP\_STATE** |
| 14 | John | 7272826385 | UP |
| 14 | John | 9064738238 | UP |
| 20 | Harry | 8574783832 | Bihar |
| 12 | Sam | 7390372389 | Punjab |
| 12 | Sam | 8589830302 | Punjab |

# 3. Second Normal Form (2NF)

* In the 2NF, relational must be in 1NF.
* In the second normal form, all non-key attributes are fully functional dependent on the primary key

**Example:** Let's assume, a school can store the data of teachers and the subjects they teach. In a school, a teacher can teach more than one subject.

**TEACHER table**

|  |  |  |
| --- | --- | --- |
| **TEACHER\_ID** | **SUBJECT** | **TEACHER\_AGE** |
| 25 | Chemistry | 30 |
| 25 | Biology | 30 |
| 47 | English | 35 |
| 83 | Math | 38 |
| 83 | Computer | 38 |

In the given table, non-prime attribute TEACHER\_AGE is dependent on TEACHER\_ID which is a proper subset of a candidate key. That's why it violates the rule for 2NF.

To convert the given table into 2NF, we decompose it into two tables:

**TEACHER\_DETAIL table:**

|  |  |
| --- | --- |
| **TEACHER\_ID** | **TEACHER\_AGE** |
| 25 | 30 |
| 47 | 35 |
| 83 | 38 |

**TEACHER\_SUBJECT table:**

|  |  |
| --- | --- |
| **TEACHER\_ID** | **SUBJECT** |
| 25 | Chemistry |
| 25 | Biology |
| 47 | English |
| 83 | Math |
| 83 | Computer |

# 4. Third Normal Form (3NF)

* A relation will be in 3NF if it is in 2NF and not contain any transitive partial dependency.
* 3NF is used to reduce the data duplication. It is also used to achieve the data integrity.
* If there is no transitive dependency for non-prime attributes, then the relation must be in third normal form.

A relation is in third normal form if it holds atleast one of the following conditions for every non-trivial function dependency X → Y.

1. X is a super key.
2. Y is a prime attribute, i.e., each element of Y is part of some candidate key.

**Example:**

**EMPLOYEE\_DETAIL table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_ZIP** | **EMP\_STATE** | **EMP\_CITY** |
| 222 | Harry | 201010 | UP | Noida |
| 333 | Stephan | 02228 | US | Boston |
| 444 | Lan | 60007 | US | Chicago |
| 555 | Katharine | 06389 | UK | Norwich |
| 666 | John | 462007 | MP | Bhopal |

**Super key in the table above:**

* 1. {EMP\_ID}, {EMP\_ID, EMP\_NAME}, {EMP\_ID, EMP\_NAME, EMP\_ZIP}....so on

**Candidate key:** {EMP\_ID}

**Non-prime attributes:** In the given table, all attributes except EMP\_ID are non-prime.

Here, EMP\_STATE & EMP\_CITY dependent on EMP\_ZIP and EMP\_ZIP dependent on EMP\_ID. The non-prime attributes (EMP\_STATE, EMP\_CITY) transitively dependent on super key(EMP\_ID). It violates the rule of third normal form.

That's why we need to move the EMP\_CITY and EMP\_STATE to the new <EMPLOYEE\_ZIP> table, with EMP\_ZIP as a Primary key.

**EMPLOYEE table:**

|  |  |  |
| --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_ZIP** |
| 222 | Harry | 201010 |
| 333 | Stephan | 02228 |
| 444 | Lan | 60007 |
| 555 | Katharine | 06389 |
| 666 | John | 462007 |

**EMPLOYEE\_ZIP table:**

|  |  |  |
| --- | --- | --- |
| **EMP\_ZIP** | **EMP\_STATE** | **EMP\_CITY** |
| 201010 | UP | Noida |
| 02228 | US | Boston |
| 60007 | US | Chicago |
| 06389 | UK | Norwich |
| 462007 | MP | Bhopal |

# 5. Boyce Codd normal form (BCNF)

* BCNF is the advance version of 3NF. It is stricter than 3NF.
* A table is in BCNF if every functional dependency X → Y, X is the super key of the table.
* For BCNF, the table should be in 3NF, and for every FD, LHS is super key.

**Example:** Let's assume there is a company where employees work in more than one department.

**EMPLOYEE table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_COUNTRY** | **EMP\_DEPT** | **DEPT\_TYPE** | **EMP\_DEPT\_NO** |
| 264 | India | Designing | D394 | 283 |
| 264 | India | Testing | D394 | 300 |
| 364 | UK | Stores | D283 | 232 |
| 364 | UK | Developing | D283 | 549 |

**In the above table Functional dependencies are as follows:**

1. EMP\_ID  →  EMP\_COUNTRY
2. EMP\_DEPT  →   {DEPT\_TYPE, EMP\_DEPT\_NO}

**Candidate key: {EMP-ID, EMP-DEPT}**

The table is not in BCNF because neither EMP\_DEPT nor EMP\_ID alone are keys.

To convert the given table into BCNF, we decompose it into three tables:

**EMP\_COUNTRY table:**

|  |  |
| --- | --- |
| **EMP\_ID** | **EMP\_COUNTRY** |
| 264 | India |
| 264 | India |

**EMP\_DEPT table:**

|  |  |  |
| --- | --- | --- |
| **EMP\_DEPT** | **DEPT\_TYPE** | **EMP\_DEPT\_NO** |
| Designing | D394 | 283 |
| Testing | D394 | 300 |
| Stores | D283 | 232 |
| Developing | D283 | 549 |

**EMP\_DEPT\_MAPPING table:**

|  |  |
| --- | --- |
| **EMP\_ID** | **EMP\_DEPT** |
| D394 | 283 |
| D394 | 300 |
| D283 | 232 |
| D283 | 549 |

**Functional dependencies:**

1. EMP\_ID   →    EMP\_COUNTRY
2. EMP\_DEPT   →   {DEPT\_TYPE, EMP\_DEPT\_NO}

**Candidate keys:**

**For the first table:** EMP\_ID  
**For the second table:** EMP\_DEPT  
**For the third table:** {EMP\_ID, EMP\_DEPT}

Now, this is in BCNF because left side part of both the functional dependencies is a key.

# 2.Transaction in DBMS

When the data of users is stored in a database, that data needs to be accessed and modified from time to time. This task should be performed with a specified set of rules and in a systematic way to maintain the consistency and integrity of the data present in a database. In DBMS, this task is called a transaction. It is similar to a bank transaction, where the user requests to withdraw some amount of money from his account. Subsequently, several operations take place such as fetching the user’s balance from the database, subtracting the desired amount from it, and updating the user’s account balance. This series of operations can be called a transaction. Transactions are very common in DBMS. In this article, we will discuss what a transaction means, various operations of transactions, transaction states, and properties of transactions in DBMS.

## What does a Transaction mean in DBMS?

Transaction in Database Management Systems (DBMS) can be defined as a set of logically related operations. It is the result of a request made by the user to access the contents of the database and perform operations on it. It consists of various operations and has various states in its completion journey. It also has some specific properties that must be followed to keep the database consistent.

## Operations of Transaction

A user can make different types of requests to access and modify the contents of a database. So, we have different types of operations relating to a transaction. They are discussed as follows:

### i) Read(X)

A read operation is used to read the value of X from the database and store it in a buffer in the main memory for further actions such as displaying that value. Such an operation is performed when a user wishes just to see any content of the database and not make any changes to it. For example, when a user wants to check his/her account’s balance, a read operation would be performed on user’s account balance from the database.

### ii) Write(X)

A write operation is used to write the value to the database from the buffer in the main memory. For a write operation to be performed, first a read operation is performed to bring its value in buffer, and then some changes are made to it, e.g. some set of arithmetic operations are performed on it according to the user’s request, then to store the modified value back in the database, a write operation is performed. For example, when a user requests to withdraw some money from his account, his account balance is fetched from the database using a read operation, then the amount to be deducted from the account is subtracted from this value, and then the obtained value is stored back in the database using a write operation.

### iii) Commit

This operation in transactions is used to maintain integrity in the database. Due to some failure of power, hardware, or software, etc., a transaction might get interrupted before all its operations are completed. This may cause ambiguity in the database, i.e. it might get inconsistent before and after the transaction. To ensure that further operations of any other transaction are performed only after work of the current transaction is done, a commit operation is performed to the changes made by a transaction permanently to the database.

### iv) Rollback

This operation is performed to bring the database to the last saved state when any transaction is interrupted in between due to any power, hardware, or software failure. In simple words, it can be said that a rollback operation does undo the operations of transactions that were performed before its interruption to achieve a safe state of the database and avoid any kind of ambiguity or inconsistency.

## Transaction Schedules

When multiple transaction requests are made at the same time, we need to decide their order of execution. Thus, a transaction schedule can be defined as a chronological order of execution of multiple transactions. There are broadly two types of transaction schedules discussed as follows,

### i) Serial Schedule

In this kind of schedule, when multiple transactions are to be executed, they are executed serially, i.e. at one time only one transaction is executed while others wait for the execution of the current transaction to be completed. This ensures consistency in the database as transactions do not execute simultaneously. But, it increases the waiting time of the transactions in the queue, which in turn lowers the throughput of the system, i.e. number of transactions executed per time. To improve the throughput of the system, another kind of schedule are used which has some more strict rules which help the database to remain consistent even when transactions execute simultaneously.

### ii) Non-Serial Schedule

To reduce the waiting time of transactions in the waiting queue and improve the system efficiency, we use nonserial schedules which allow multiple transactions to start before a transaction is completely executed. This may sometimes result in inconsistency and errors in database operation. So, these errors are handled with specific algorithms to maintain the consistency of the database and improve CPU throughput as well. Serial Schedules are also sometimes referred to as parallel schedules as transactions execute in parallel in this kind of schedules.

### Serializable

Serializability in DBMS is the property of a nonserial schedule that determines whether it would maintain the database consistency or not. The nonserial schedule which ensures that the database would be consistent after the transactions are executed in the order determined by that schedule is said to be Serializable Schedules. The serial schedules always maintain database consistency as a transaction starts only when the execution of the other transaction has been completed under it. Thus, serial schedules are always serializable.

A transaction is a series of operations, so various states occur in its completion journey. They are discussed as follows:

### i) Active

It is the first stage of any transaction when it has begun to execute. The execution of the transaction takes place in this state. Operations such as insertion, deletion, or updation are performed during this state. During this state, the data records are under manipulation and they are not saved to the database, rather they remain somewhere in a buffer in the main memory.

### ii) Partially Committed

This state of transaction is achieved when it has completed most of the operations and is executing its final operation. It can be a signal to the commit operation, as after the final operation of the transaction completes its execution, the data has to be saved to the database through the commit operation. If some kind of error occurs during this state, the transaction goes into a failed state, else it goes into the Committed state.

### iii) Commited

This state of transaction is achieved when all the transaction-related operations have been executed successfully along with the Commit operation, i.e. data is saved into the database after the required manipulations in this state. This marks the successful completion of a transaction.

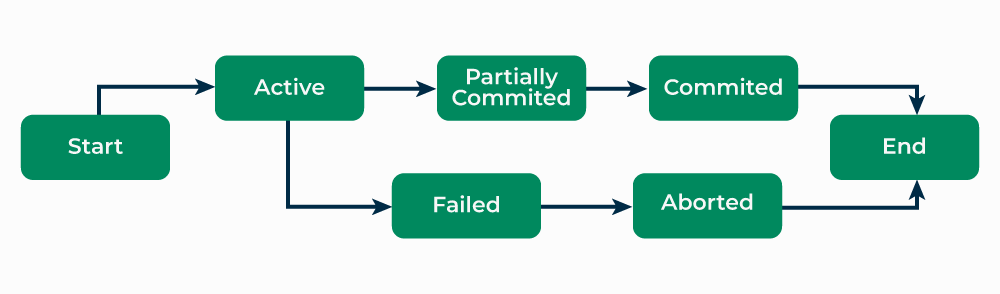
### iv) Failed

If any of the transaction-related operations cause an error during the active or partially committed state, further execution of the transaction is stopped and it is brought into a failed state. Here, the database recovery system makes sure that the database is in a consistent state.

### v) Aborted

If the error is not resolved in the failed state, then the transaction is aborted and a rollback operation is performed to bring database to the the last saved consistent state. When the transaction is aborted, the database recovery module either restarts the transaction or kills it.

The illustration below shows the various states that a transaction may encounter in its completion journey.



*Transaction in DBMS*

## Properties of Transaction

As transactions deal with accessing and modifying the contents of the database, they must have some basic properties which help maintain the consistency and integrity of the database before and after the transaction. Transactions follow 4 properties, namely, Atomicity, Consistency, Isolation, and Durability. Generally, these are referred to as ACID properties of transactions in DBMS. ACID is the acronym used for transaction properties. A brief description of each property of the transaction is as follows.

### i) Atomicity

This property ensures that either all operations of a transaction are executed or it is aborted. In any case, a transaction can never be completed partially. Each transaction is treated as a single unit (like an atom). Atomicity is achieved through commit and rollback operations, i.e. changes are made to the database only if all operations related to a transaction are completed, and if it gets interrupted, any changes made are rolled back using rollback operation to bring the database to its last saved state.

### ii) Consistency

This property of a transaction keeps the database consistent before and after a transaction is completed. Execution of any transaction must ensure that after its execution, the database is either in its prior stable state or a new stable state. In other words, the result of a transaction should be the transformation of a database from one consistent state to another consistent state. Consistency, here means, that the changes made in the database are a result of logical operations only which the user desired to perform and there is not any ambiguity.

### iii) Isolation

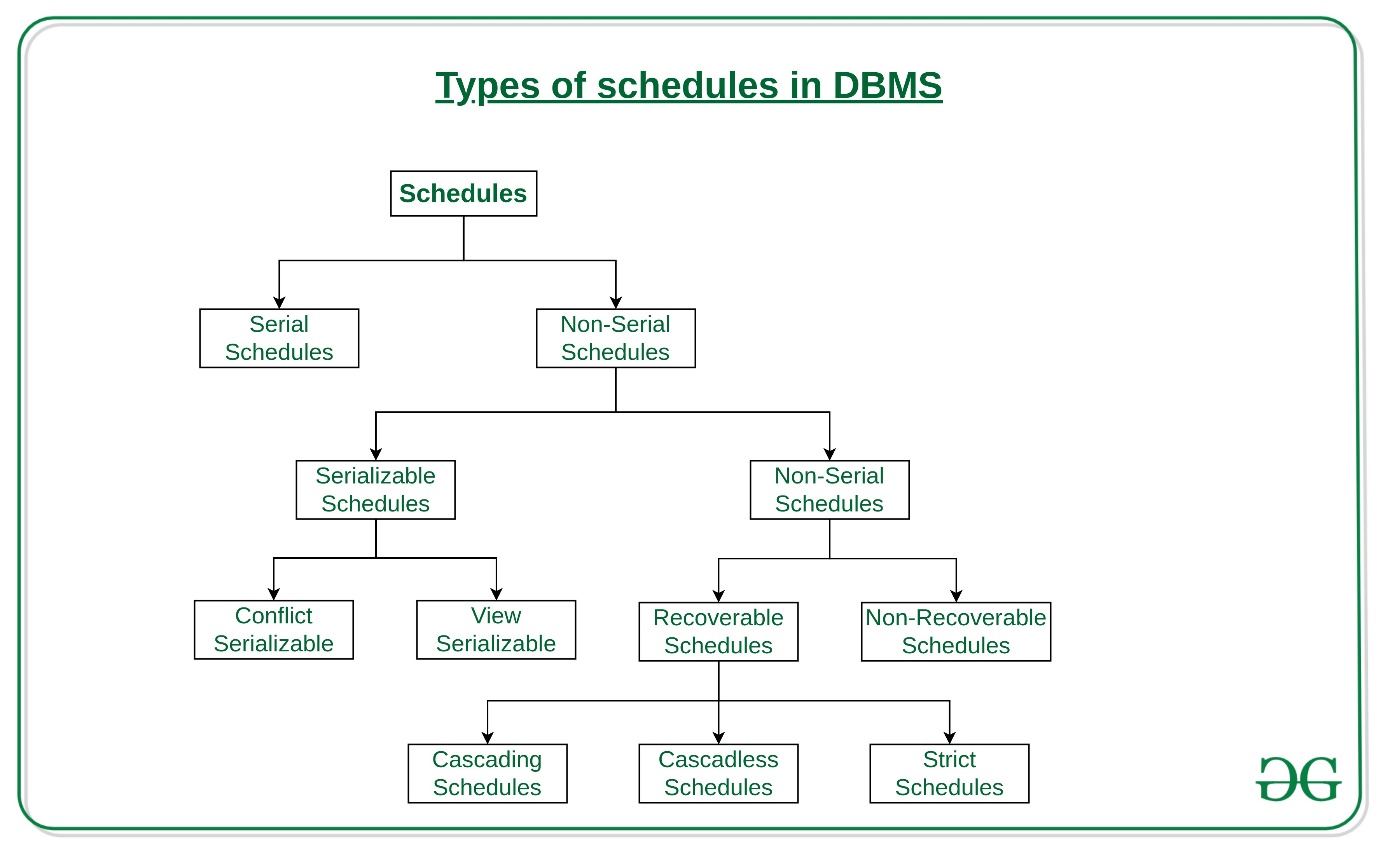
This property states that two transactions must not interfere with each other, i.e. if some data is used by a transaction for its execution, then any other transaction can not concurrently access that data until the first transaction has completed. It ensures that the integrity of the database is maintained and we don’t get any ambiguous values. Thus, any two transactions are isolated from each other. This property is enforced by the concurrency control subsystem of DBMS.

### iv) Durability

This property ensures that the changes made to the database after a transaction is completely executed, are durable. It indicates that permanent changes are made by the successful execution of a transaction. In the event of any system failures or crashes, the consistent state achieved after the completion of a transaction remains intact. The recovery subsystem of DBMS is responsible for enforcing this property.

# 3.Types of Schedules in DBMS

Schedule, as the name suggests, is a process of lining the transactions and executing them one by one. When there are multiple transactions that are running in a concurrent manner and the order of operation is needed to be set so that the operations do not overlap each other, Scheduling is brought into play and the transactions are timed accordingly.



1. **Serial Schedules:**  
   Schedules in which the transactions are executed non-interleaved, i.e., a serial schedule is one in which no transaction starts until a running transaction has ended are called serial schedules.

**Example:** Consider the following schedule involving two transactions T1 and T2.

| **T1** | **T2** |
| --- | --- |
| R(A) |  |
| W(A) |  |
| R(B) |  |
|  | W(B) |
|  | R(A) |
|  | R(B) |

where R(A) denotes that a read operation is performed on some data item ‘A’  
This is a serial schedule since the transactions perform serially in the order T1 —> T2

1. **Non-Serial Schedule:**  
   This is a type of Scheduling where the operations of multiple transactions are interleaved. This might lead to a rise in the concurrency problem. The transactions are executed in a non-serial manner, keeping the end result correct and same as the serial schedule. Unlike the serial schedule where one transaction must wait for another to complete all its operation, in the non-serial schedule, the other transaction proceeds without waiting for the previous transaction to complete. This sort of schedule does not provide any benefit of the concurrent transaction. It can be of two types namely, Serializable and Non-Serializable Schedule.

The Non-Serial Schedule can be divided further into Serializable and Non-Serializable.

* 1. **Serializable:**  
     This is used to maintain the consistency of the database. It is mainly used in the Non-Serial scheduling to verify whether the scheduling will lead to any inconsistency or not. On the other hand, a serial schedule does not need the serializability because it follows a transaction only when the previous transaction is complete. The non-serial schedule is said to be in a serializable schedule only when it is equivalent to the serial schedules, for an n number of transactions. Since concurrency is allowed in this case thus, multiple transactions can execute concurrently. A serializable schedule helps in improving both resource utilization and CPU throughput. These are of two types:
     1. [Conflict Serializable:](https://www.geeksforgeeks.org/conflict-serializability/)  
        A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations. Two operations are said to be conflicting if all conditions satisfy:
        + They belong to different transactions
        + They operate on the same data item
        + At Least one of them is a write operation
     2. [View Serializable:](https://www.geeksforgeeks.org/dbms-how-to-test-two-schedule-are-view-equal-or-not-2/)  
        A Schedule is called view serializable if it is view equal to a serial schedule (no overlapping transactions). A conflict schedule is a view serializable but if the serializability contains blind writes, then the view serializable does not conflict serializable.
  2. **Non-Serializable:**  
     The non-serializable schedule is divided into two types, Recoverable and Non-recoverable Schedule.
     1. [Recoverable Schedule:](https://www.geeksforgeeks.org/recoverability-in-dbms/)  
        Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules. In other words, if some transaction Tj is reading value updated or written by some other transaction Ti, then the commit of Tj must occur after the commit of Ti.

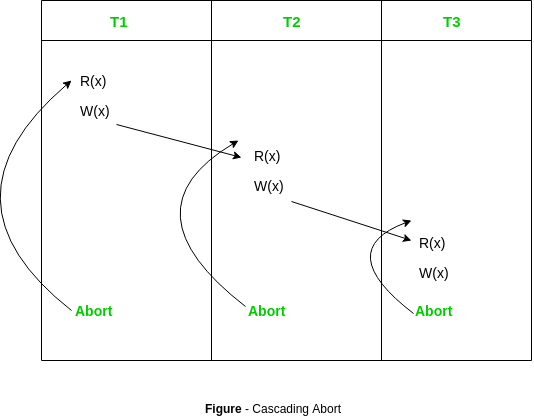
**Example –** Consider the following schedule involving two transactions T1 and T2.

| **T1** | **T2** |
| --- | --- |
| R(A) |  |
| W(A) |  |
|  | W(A) |
|  | R(A) |
| commit |  |
|  | commit |

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

There can be three types of recoverable schedule:

* + - * **Cascading Schedule:**

Also called Avoids cascading aborts/rollbacks (ACA). When there is a failure in one transaction and this leads to the rolling back or aborting other dependent transactions, then such scheduling is referred to as Cascading rollback or cascading abort. Example:  


* + - * [Cascadeless Schedule:](https://www.geeksforgeeks.org/cascadeless-in-dbms/)  
        Schedules in which transactions read values only after all transactions whose changes they are going to read commit are called cascadeless schedules. Avoids that a single transaction abort leads to a series of transaction rollbacks. A strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule.

In other words, if some transaction Tj wants to read value updated or written by some other transaction Ti, then the commit of Tj must read it after the commit of Ti.

**Example:** Consider the following schedule involving two transactions T1 and T2.

| **T1** | **T2** |
| --- | --- |
| R(A) |  |
| W(A) |  |
|  | W(A) |
| commit |  |
|  | R(A) |
|  | commit |

This schedule is cascadeless. Since the updated value of **A** is read by T2 only after the updating transaction i.e. T1 commits.

**Example:** Consider the following schedule involving two transactions T1 and T2.

| **T1** | **T2** |
| --- | --- |
| R(A) |  |
| W(A) |  |
|  | R(A) |
|  | W(A) |
| abort |  |
|  | abort |

It is a recoverable schedule but it does not avoid cascading aborts. It can be seen that if T1 aborts, T2 will have to be aborted too in order to maintain the correctness of the schedule as T2 has already read the uncommitted value written by T1.

* + - * **Strict Schedule:**  
        A schedule is strict if for any two transactions Ti, 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 updated or written value of Ti only after Ti commits/aborts.

**Example:** Consider the following schedule involving two transactions T1 and T2.

| **T1** | **T2** |
| --- | --- |
| R(A) |  |
|  | R(A) |
| W(A) |  |
| commit |  |
|  | W(A) |
|  | R(A) |
|  | commit |

This is a strict schedule since T2 reads and writes A which is written by T1 only after the commit of T1.

* + 1. **Non-Recoverable Schedule:**  
       **Example:** Consider the following schedule involving two transactions T1 and T2.

| **T1** | **T2** |
| --- | --- |
| R(A) |  |
| W(A) |  |
|  | W(A) |
|  | R(A) |
|  | commit |
| abort |  |

* + 1. T2 read the value of A written by T1, and committed. T1 later aborted, therefore the value read by T2 is wrong, but since T2 committed, this schedule is **non-recoverable**.

**Note –** It can be seen that:

1. Cascadeless schedules are stricter than recoverable schedules or are a subset of recoverable schedules.
2. Strict schedules are stricter than cascadeless schedules or are a subset of cascadeless schedules.
3. Serial schedules satisfy constraints of all recoverable, cascadeless and strict schedules and hence is a subset of strict schedules.

# 4.Conflict Serializability in DBMS

As discussed in Concurrency control, serial schedules have less resource utilization and low throughput. To improve it, two or more transactions are run concurrently. However, concurrency of transactions may lead to inconsistency in the database. To avoid this, we need to check whether these concurrent schedules are serializable or not.

## **Conflict Serializable**

Concurrency serializability, also known as conflict serializability, is a type of concurrency control that guarantees that the outcome of concurrent transactions is the same as if the transactions were executed consecutively.

**Conflict serializable schedules:** A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.

**Non-conflicting operations:**When two operations operate on separate data items or the same data item but at least one of them is a read operation, they are said to be non-conflicting.

## **Conflicting Operations**

Two operations are said to be conflicting if all conditions are satisfied:

* They belong to different transactions
* They operate on the same data item
* At Least one of them is a write operation

**Example:**

* **Conflicting** operations pair (R1(A), W2(A)) because they belong to two different transactions on the same data item A and one of them is a write operation.
* Similarly, (W1(A), W2(A)) and (W1(A), R2(A)) pairs are also **conflicting**.
* On the other hand, the (R1(A), W2(B)) pair is **non-conflicting** because they operate on different data items.
* Similarly, ((W1(A), W2(B)) pair is **non-conflicting.**

Consider the following schedule:

S1: R1(A), W1(A), R2(A), W2(A), R1(B), W1(B), R2(B), W2(B)

If Oi and Oj are two operations in a transaction and Oi< Oj (Oi is executed before Oj), same order will follow in the schedule as well. Using this property, we can get two transactions of schedule S1:

T1: R1(A), W1(A), R1(B), W1(B)  
T2: R2(A), W2(A), R2(B), W2(B)

**Possible Serial Schedules are: T1->T2 or T2->T1**

-> **Swapping non-conflicting operation**s R2(A) and R1(B) in S1, the schedule becomes,

**S11:** R1(A), W1(A), R1(B), **W2(A),** R2(A), **W1(B),** R2(B), W2(B)

-> Similarly, **swapping non-conflicting operations** W2(A) and W1(B) in S11, the schedule becomes,

**S12:** R1(A), W1(A), R1(B), W1(B), R2(A), W2(A), R2(B), W2(B)

S12 is a serial schedule in which all operations of T1 are performed before starting any operation of T2. Since S has been transformed into a serial schedule S12 by swapping non-conflicting operations of S1, S1 is conflict serializable.

Let us take another Schedule:

S2: R2(A), W2(A), R1(A), W1(A), R1(B), W1(B), R2(B), W2(B)

Two transactions will be:

T1: R1(A), W1(A), R1(B), W1(B)  
T2: R2(A), W2(A), R2(B), W2(B)

**Possible Serial Schedules are: T1->T2 or T2->T1**

Original Schedule is as:

**S2:** R2(A), W2(A), **R1(A),** W1(A), R1(B), W1(B), **R2(B),** W2(B)

Swapping non-conflicting operations R1(A) and R2(B) in S2, the schedule becomes,

**S21:** R2(A), W2(A), R2(B), **W1(A),** R1(B), W1(B), R1(A), **W2(B)**

Similarly, swapping non-conflicting operations W1(A) and W2(B) in S21, the schedule becomes,

**S22:** R2(A), W2(A), R2(B), W2(B), R1(B), W1(B), R1(A), W1(A)

In schedule S22, all operations of T2 are performed first, but operations of T1 are not in order (order should be R1(A), W1(A), R1(B), W1(B)). So S2 is not conflict serializable.

## **Conflict Equivalent**

Two schedules are said to be conflict equivalent when one can be transformed to another by swapping non-conflicting operations. In the example discussed above, S11 is conflict equivalent to S1 (S1 can be converted to S11 by swapping non-conflicting operations). Similarly, S11 is conflict equivalent to S12, and so on.

**Note 1:** Although S2 is not conflict serializable, still it is conflict equivalent to S21 and S21 because S2 can be converted to S21 and S22 by swapping non-conflicting operations.

**Note 2:** The schedule which is conflict serializable is always conflict equivalent to one of the serial schedule. S1 schedule discussed above (which is conflict serializable) is equivalent to the serial schedule (T1->T2).

# 5.Condition of schedules to View-equivalent

In a database system, a schedule is a sequence of operations (such as read and write operations) performed by transactions in the system. Two schedules are view-equivalent if they produce the same set of results when executed against the same database state.

A schedule S1 is said to be view-equivalent to a schedule S2 if and only if:

The order of any two conflicting operations in S1 is the same as the order of those operations in S2. A conflicting operation is an operation that accesses the same data item as another operation and at least one of the operations is a write operation.

The order of any two non-conflicting operations can be interchanged without changing the results produced by the schedules.

In other words, two schedules are view-equivalent if they produce the same results regardless of the order in which non-conflicting operations are executed, and the order of conflicting operations is the same in both schedules.

The condition of schedules to be view-equivalent is important because it determines whether two schedules can be considered equivalent for purposes such as concurrency control, recovery, and replication. If two schedules are view-equivalent, it means that they are interchangeable in terms of their results, and the system can choose to execute either one without affecting the consistency or correctness of the database.

View-equivalence is a weaker condition than conflict serializability, which requires that two schedules be equivalent in terms of the order in which transactions are executed. View equivalence allows for more concurrency and flexibility in scheduling, as long as the results produced by the schedules are the same.

Two schedules S1 and S2 are said to be view-equivalent if the below conditions are satisfied :

**1) Initial Read:**If a transaction T1 reads data item A from the database in S1 then in S2 also T1 should read A from database.

T1 T2 T3

-------------------

R(A)

W(A)

R(A)

R(B)

Transaction T2 is reading A from the database.

**2) Updated Read:**If Ti is reading A which is updated by Tj in S1 then in S2 also Ti should read A which is updated by Tj.

T1 T2 T3 T1 T2 T3

------------------- ----------------

W(A) W(A)

W(A) R(A)

R(A) W(A)

Above two schedules are not view-equivalent as in S1 :T3 is reading A updated by T2, in S2 T3 is reading A updated by T1.

**3) Final Write operation:**If a transaction T1 updated A at last in S1, then in S2 also T1 should perform final write operations.

T1 T2 T1 T2

------------ ---------------

R(A) R(A)

W(A) W(A)

W(A) W(A)

Above two schedules are not view-equivalent as Final write operation in S1 is done by T1 while in S2 done by T2.

**View Serializability:**A Schedule is called view serializable if it is view equal to a serial schedule (no overlapping transactions).

# 6.Recoverability in DBMS

**Recoverability** is a property of database systems that ensures that, in the event of a failure or error, the system can recover the database to a consistent state. Recoverability guarantees that all committed transactions are durable and that their effects are permanently stored in the database, while the effects of uncommitted transactions are undone to maintain data consistency.

The recoverability property is enforced through the use of transaction logs, which record all changes made to the database during transaction processing. When a failure occurs, the system uses the log to recover the database to a consistent state, which involves either undoing the effects of uncommitted transactions or redoing the effects of committed transactions.

#### There are several levels of recoverability that can be supported by a database system:

**No-undo logging:**This level of recoverability only guarantees that committed transactions are durable, but does not provide the ability to undo the effects of uncommitted transactions.

**Undo logging:** This level of recoverability provides the ability to undo the effects of uncommitted transactions but may result in the loss of updates made by committed transactions that occur after the failed transaction.

**Redo logging:** This level of recoverability provides the ability to redo the effects of committed transactions, ensuring that all committed updates are durable and can be recovered in the event of failure.

**Undo-redo logging:**This level of recoverability provides both undo and redo capabilities, ensuring that the system can recover to a consistent state regardless of whether a transaction has been committed or not.

In addition to these levels of recoverability, database systems may also use techniques such as checkpointing and shadow paging to improve recovery performance and reduce the overhead associated with logging.

Overall, recoverability is a crucial property of database systems, as it ensures that data is consistent and durable even in the event of failures or errors. It is important for database administrators to understand the level of recoverability provided by their system and to configure it appropriately to meet their application’s requirements.

Prerequisite:

* [Introduction to Concurrency Control](https://www.geeksforgeeks.org/concurrency-control-in-dbms/)
* [Types of Schedules](https://www.geeksforgeeks.org/dbms-concurrency-control-types-of-schedules/)

As discussed, a transaction may not execute completely due to hardware failure, system crash or software issues. In that case, we have to roll back the failed transaction. But some other transactions may also have used values produced by the failed transaction. So we have to roll back those transactions as well.

**Recoverable Schedules:**

* Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules. In other words, if some transaction Tj is reading value updated or written by some other transaction Ti, then the commit of Tj must occur after the commit of Ti.

**Example 1:**

S1: R1(x), **W1(x)**, R2(x), R1(y), R2(y),

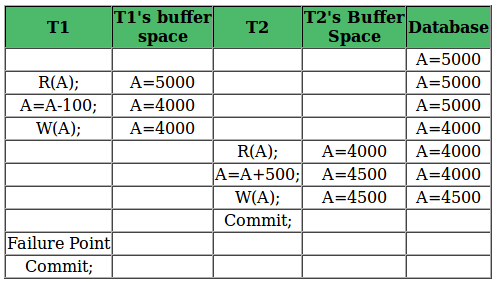
**W2(x)**, W1(y), **C1**, **C2**;

Given schedule follows order of **Ti->Tj => C1->C2**. Transaction T1 is executed before T2 hence there is no chances of conflict occur. R1(x) appears before W1(x) and transaction T1 is committed before T2 i.e. completion of first transaction performed first update on data item x, hence given schedule is recoverable.

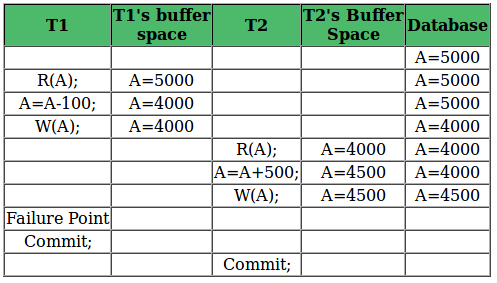
**Example 2:** Consider the following schedule involving two transactions T1 and T2.

| **T1** | **T2** |
| --- | --- |
| R(A) |  |
| W(A) |  |
|  | W(A) |
|  | R(A) |
| commit |  |
|  | commit |

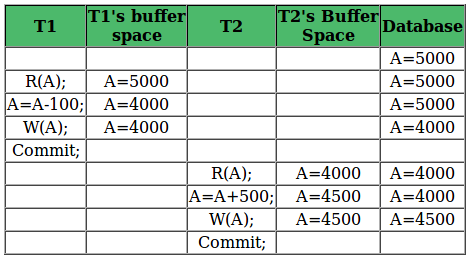
This is a recoverable schedule since T1 commits before T2, that makes the value read by T2 correct.

**Irrecoverable Schedule:**The table below shows a schedule with two transactions, T1 reads and writes A and that value is read and written by T2. T2 commits. But later on, T1 fails. So we have to rollback T1. Since T2 has read the value written by T1, it should also be rollbacked. But we have already committed that. So this schedule is irrecoverable schedule. When Tj is reading the value updated by Ti and Tj is committed before committing of Ti, the schedule will be irrecoverable.[](https://media.geeksforgeeks.org/wp-content/uploads/scheduleDBMS.png)

**Recoverable with Cascading Rollback:**The table below shows a schedule with two transactions, T1 reads and writes A and that value is read and written by T2. But later on, T1 fails. So we have to rollback T1. Since T2 has read the value written by T1, it should also be rollbacked. As it has not committed, we can rollback T2 as well. So it is recoverable with cascading rollback. Therefore, if Tj is reading value updated by Ti and commit of Tj is delayed till commit of Ti, the schedule is called recoverable with cascading rollback.

[](https://media.geeksforgeeks.org/wp-content/uploads/schedule5.png)

**Cascadeless Recoverable Rollback:**The table below shows a schedule with two transactions, T1 reads and writes A and commits and that value is read by T2. But if T1 fails before commit, no other transaction has read its value, so there is no need to rollback other transaction. So this is a Cascadeless recoverable schedule. So, if Tj reads value updated by Ti only after Ti is committed, the schedule will be cascadeless recoverable.

[](https://media.geeksforgeeks.org/wp-content/uploads/schedult3.png)

# 7.Database Recovery Techniques in DBMS

Database Systems like any other computer system, are subject to failures but the data stored in them must be available as and when required. When a database fails it must possess the facilities for fast recovery. It must also have atomicity i.e. either transactions are completed successfully and committed (the effect is recorded permanently in the database) or the transaction should have no effect on the database.

## Types of Recovery Techniques in DBMS

Database recovery techniques are used in database management systems (DBMS) to restore a database to a consistent state after a failure or error has occurred. The main goal of recovery techniques is to ensure data integrity and consistency and prevent data loss.

There are mainly two types of recovery techniques used in DBMS

* **Rollback/Undo Recovery Technique**
* **Commit/Redo Recovery Technique**

### **Rollback/Undo Recovery Technique**

The rollback/undo recovery technique is based on the principle of backing out or undoing the effects of a transaction that has not been completed successfully due to a system failure or error. This technique is accomplished by undoing the changes made by the transaction using the log records stored in the transaction log. The transaction log contains a record of all the transactions that have been performed on the database. The system uses the log records to undo the changes made by the failed transaction and restore the database to its previous state.

### **Commit/Redo Recovery Technique**

The commit/redo recovery technique is based on the principle of reapplying the changes made by a transaction that has been completed successfully to the database. This technique is accomplished by using the log records stored in the transaction log to redo the changes made by the transaction that was in progress at the time of the failure or error. The system uses the log records to reapply the changes made by the transaction and restore the database to its most recent consistent state.

In addition to these two techniques, there is also a third technique called [checkpoint recovery](https://www.geeksforgeeks.org/checkpoints-in-dbms/).

**Checkpoint Recovery**is a technique used to reduce the recovery time by periodically saving the state of the database in a checkpoint file. In the event of a failure, the system can use the checkpoint file to restore the database to the most recent consistent state before the failure occurred, rather than going through the entire log to recover the database.

Overall, recovery techniques are essential to ensure data consistency and availability in [Database Management System](https://www.geeksforgeeks.org/introduction-of-dbms-database-management-system-set-1/), and each technique has its own advantages and limitations that must be considered in the design of a recovery system.

## **Database Systems**

There are both automatic and non-automatic ways for both, backing up data and recovery from any failure situations. The techniques used to recover lost data due to system crashes, transaction errors, viruses, catastrophic failure, incorrect command execution, etc. are database recovery techniques. So to prevent data loss recovery techniques based on deferred updates and immediate updates or backing up data can be used. Recovery techniques are heavily dependent upon the existence of a special file known as a **system log**. It contains information about the start and end of each transaction and any updates which occur during the **transaction**. The log keeps track of all transaction operations that affect the values of database items. This information is needed to recover from transaction failure.

* **The log is kept on disk start\_transaction(T):** This log entry records that transaction T starts the execution.
* **read\_item(T, X):** This log entry records that transaction T reads the value of database item X.
* **write\_item(T, X, old\_value, new\_value):** This log entry records that transaction T changes the value of the database item X from old\_value to new\_value. The old value is sometimes known as a before an image of X, and the new value is known as an afterimage of X.
* **commit(T):** This log entry records that transaction T has completed all accesses to the database successfully and its effect can be committed (recorded permanently) to the database.
* **abort(T):**This records that transaction T has been aborted.
* **checkpoint:**A checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk. Checkpoint declares a point before which the DBMS was in a consistent state, and all the transactions were committed.

A transaction T reaches its **commit** point when all its operations that access the database have been executed successfully i.e. the transaction has reached the point at which it will not **abort** (terminate without completing). Once committed, the transaction is permanently recorded in the database. Commitment always involves writing a commit entry to the log and writing the log to disk. At the time of a system crash, the item is searched back in the log for all transactions T that have written a start\_transaction(T) entry into the log but have not written a commit(T) entry yet; these transactions may have to be rolled back to undo their effect on the database during the recovery process.

* **Undoing:** If a transaction crashes, then the recovery manager may undo transactions i.e. reverse the operations of a transaction. This involves examining a transaction for the log entry write\_item(T, x, old\_value, new\_value) and setting the value of item x in the database to old-value. There are two major techniques for recovery from non-catastrophic transaction failures: [deferred updates and immediate updates.](https://www.geeksforgeeks.org/difference-between-deferred-update-and-immediate-update/)
* **Deferred Update:** This technique does not physically update the database on disk until a transaction has reached its commit point. Before reaching commit, all transaction updates are recorded in the local transaction workspace. If a transaction fails before reaching its commit point, it will not have changed the database in any way so UNDO is not needed. It may be necessary to REDO the effect of the operations that are recorded in the local transaction workspace, because their effect may not yet have been written in the database. Hence, a deferred update is also known as the **No-undo/redo algorithm.**
* **Immediate Update:** In the immediate update, the database may be updated by some operations of a transaction before the transaction reaches its commit point. However, these operations are recorded in a log on disk before they are applied to the database, making recovery still possible. If a transaction fails to reach its commit point, the effect of its operation must be undone i.e. the transaction must be rolled back hence we require both undo and redo. This technique is known as **undo/redo algorithm.**
* **Caching/Buffering:** In this one or more disk pages that include data items to be updated are cached into main memory buffers and then updated in memory before being written back to disk. A collection of in-memory buffers called the DBMS cache is kept under the control of DBMS for holding these buffers. A directory is used to keep track of which database items are in the buffer. A dirty bit is associated with each buffer, which is 0 if the buffer is not modified else 1 if modified.
* **Shadow Paging:** It provides atomicity and durability. A directory with n entries is constructed, where the ith entry points to the ith database page on the link. When a transaction began executing the current directory is copied into a shadow directory. When a page is to be modified, a shadow page is allocated in which changes are made and when it is ready to become durable, all pages that refer to the original are updated to refer new replacement page.
* **Backward Recovery:** The term “**Rollback**” and “**UNDO**” can also refer to backward recovery. When a backup of the data is not available and previous modifications need to be undone, this technique can be helpful. With the backward recovery method, unused modifications are removed and the database is returned to its prior condition. All adjustments made during the previous traction are reversed during the backward recovery. In other words, it reprocesses valid transactions and undoes the erroneous database updates.
* **Forward Recovery:** “**Roll forward** “and “**REDO**” refers to forwarding recovery. When a database needs to be updated with all changes verified, this forward recovery technique is helpful. Some failed transactions in this database are applied to the database to roll those modifications forward. In other words, the database is restored using preserved data and valid transactions counted by their past saves.

## Backup Techniques

There are different types of Backup Techniques. Some of them are listed below.

* **Full database Backup:** In this full database including data and database, Meta information needed to restore the whole database, including full-text catalogs are backed up in a predefined time series.
* **Differential Backup:** It stores only the data changes that have occurred since the last full database backup. When some data has changed many times since the last full database backup, a differential backup stores the most recent version of the changed data. For this first, we need to restore a full database backup.
* **Transaction Log Backup:** In this, all events that have occurred in the database, like a record of every single statement executed is backed up. It is the backup of transaction log entries and contains all transactions that had happened to the database. Through this, the database can be recovered to a specific point in time. It is even possible to perform a backup from a transaction log if the data files are destroyed and not even a single committed transaction is lost.