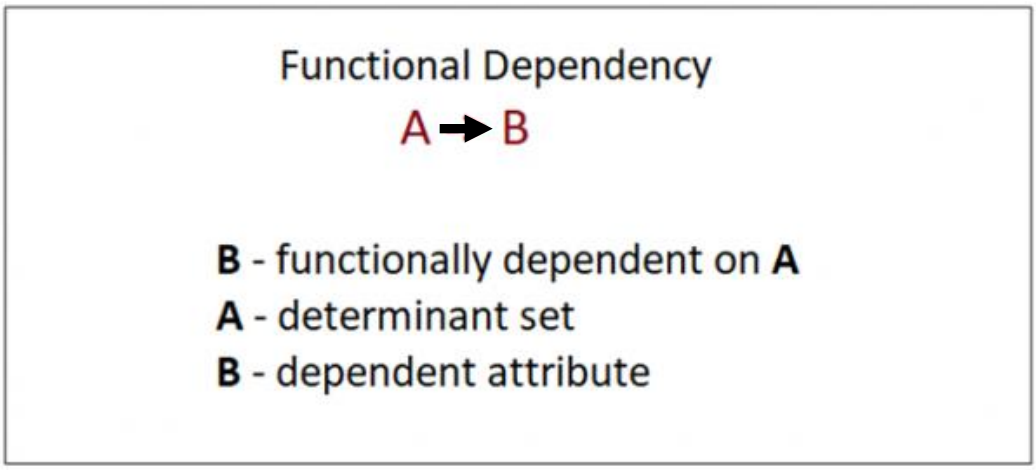
Unit-3

Functional Dependency

Functional Dependency (FD) determines the relation of one attribute to another attribute in a database management system (DBMS) system. Functional dependency helps us to maintain the quality of data in the database. A functional dependency is denoted by an arrow →. The functional dependency of B on A is represented by A → B. Functional Dependency plays a very important role to find the difference between good and bad database design.



Example:

Employee number	Employee Name	Salary	City
101	Mohan	50000	Moradabad
201	Geeta	38000	Rampur
301	Shyam	25000	Amroha

In this example, if we know the value of Employee number, we can obtain Employee Name, city, salary, etc. By this, we can say that the city, Employee Name, and salary are functionally depended on Employee number.

{Employee number → Employee Name, Salary, City}

Key terms

Here, are some key terms for functional dependency:

Key Terms	Description
Axiom	Axioms are a set of inference rules used to infer all the functional dependencies on a relational database.
Decomposition	It is a rule that suggests if we have a table that appears to contain two entities which are determined by the same primary key then we should consider breaking them up into two different tables.
Dependent	It is displayed on the right side of the functional dependency diagram.
Determinant	It is displayed on the left side of the functional dependency Diagram.
Union	It suggests that if two tables are separate, and the PK is the same, we should consider putting them together.

Rules of Functional Dependencies

Below given are the three most important rules for Functional Dependency:

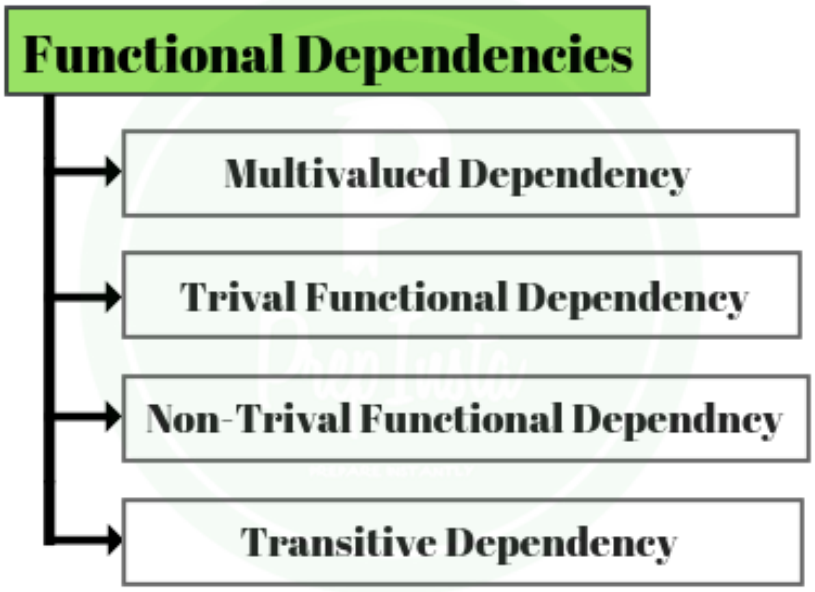
- **Reflexive rule:** –If X is a set of attributes and Y is subset of X, then X holds a value of Y.

  Ex: X= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

  Y= {1, 2, 3, 4, 5}
- **Augmentation rule:** When a → b holds, and c is attribute set, then ac → bc also holds. That is adding attributes which do not change the basic dependencies.

- **Transitivity rule:** This rule is very much similar to the transitive rule in algebra if x → y holds and y → z holds, then x → z also holds.

Types of Functional Dependencies



1. **Multivalued dependency in DBMS**

Multivalued dependency occurs in the situation where there are multiple independent multivalued attributes in a single table.

- Multivalued dependency occurs when two attributes in a table are independent of each other but, both depend on a third attribute.
- A multivalued dependency consists of at least two attributes that are dependent on a third attribute that's why it always requires at least three attributes.
- It is represented by double arrow →→

  For example:

  P→→Q

  P→→R

Example

Let us see an example

StudentName	CourseDiscipline	Activities
Amit	BCA	Singing
Amit	BCA	Dancing
Yuvraj	B.Tech	Cricket
Akash	MCA	Dancing
Akash	MCA	Cricket
Akash	MCA	Singing

In the above table, we can see Students **Amit** and **Akash** have interest in more than one activity. This is multivalued dependency because **CourseDiscipline** of a student are independent of Activities, but are dependent on the student.

Therefore, multivalued dependency –

StudentName →→ CourseDiscipline

StudentName →→ Activities

2. <u>Trivial Functional dependency:</u>

The Trivial dependency is a set of attributes which are called a trivial if the set of attributes are included in that attribute. If a functional dependency (FD) X → Y holds, where Y is a subset of X, then it is called a trivial FD.

So, X→Y is a trivial functional dependency if Y is a subset of X.

For example:

Emp_id	Emp_name
AS555	Ram
AS811	Krishna
AS999	Aarav

Consider this table with two columns Emp_id and Emp_name.

Emp_id →{Emp_id, Emp_name} is a trivial functional dependency as Emp_id is a subset of {Emp_id,Emp_name}.

3. Non trivial functional dependency in DBMS

Functional dependency is known as a non-trivial dependency when A→B holds true where B is not a subset of A. In a relationship, if attribute B is not a subset of attribute A, then it is considered as a non-trivial dependency.

Company	CEO	Age
Microsoft	Satya Nadella	51
Google	Sundar Pichai	46
Apple	Tim Cook	57

Example:

{Company} → {CEO} (if we know the Company, we knows the CEO name)

But CEO is not a subset of Company, and hence it's non-trivial functional dependency.

4. Transitive dependency:

A transitive is a type of functional dependency which happens when it is indirectly formed by two functional dependencies.

Example:

Company	CEO	Age
Microsoft	Satya Nadella	51
Google	Sundar Pichai	46
Alibaba	Jack Ma	54

{Company} → {CEO} (if we know the compay, we know its CEO's name)

{CEO } → {Age} If we know the CEO, we know the Age

Therefore according to the rule of transitive dependency:

{Company} → {Age} should hold, that makes sense because if we know the company name, we can know his age.

Note: We need to remember that transitive dependency can only occur in a relation of three or more attributes.

Advantages of Functional Dependency

- Functional Dependency avoids data redundancy. Therefore same data do not repeat at multiple locations in that database.
- It helps us to maintain the quality of data in the database.
- It helps us to defined meanings and constraints of databases.
- It helps us to identify bad designs.
- It helps us to find the facts regarding the database design.

Normalization

Database Normalization is a technique of organizing the data in the database. Normalization is a systematic approach of decomposing tables to eliminate data redundancy (repetition) and undesirable characteristics like Insertion, Update and Deletion Anomalies. It is a multi-step process that puts data into tabular form, removing duplicated data from the relation tables.

Normalization is used for mainly two purposes,

- Eliminating redundant (useless) data.
- Ensuring data dependencies make sense i.e data is logically stored.

The inventor of the relational model Edgar Codd proposed the theory of normalization with the introduction of the First Normal Form, and he continued to extend theory with Second and Third Normal Form. Later he joined Raymond F. Boyce to develop the theory of Boyce-Codd Normal Form (BCNF).

Anomalies in DBMS

There are three types of anomalies that occur when the database is not normalized. These are – Insertion, update and deletion anomaly. Let's take an example to understand this.

Example: Suppose a manufacturing company stores the employee details in a table named employee that has four attributes: emp_id for storing employee's id, emp_name for storing employee's name, emp_address for storing employee's address and emp_dept for storing the department details in which the employee works. At some point of time the table looks like this:

emp_id	emp_name	emp_address	emp_dept
101	Ashish	Delhi	D001
101	Ashish	Delhi	D002
123	Seema	Agra	D890
166	Ankit	Chennai	D900
166	Ankit	Chennai	D004

The above table is not normalized. We will see the problems that we face when a table is not normalized.

Update anomaly: In the above table we have two rows for employee Ashish as he belongs to two departments of the company. If we want to update the address of Ashish then we have to update the same in two rows or the data will become inconsistent. If somehow, the correct address gets updated in one department but not in other then as per the database, Ashish would be having two different addresses, which is not correct and would lead to inconsistent data.

Insert anomaly: Suppose a new employee joins the company, who is under training and currently not assigned to any department then we would not be able to insert the data into the table if emp_dept field doesn't allow nulls.

Delete anomaly: Suppose, if at a point of time the company closes the department D890 then deleting the rows that are having emp_dept as D890 would also delete the information of employee Seema since she is assigned only to this department.

To overcome these anomalies we need to normalize the data.

Types of Normal Forms

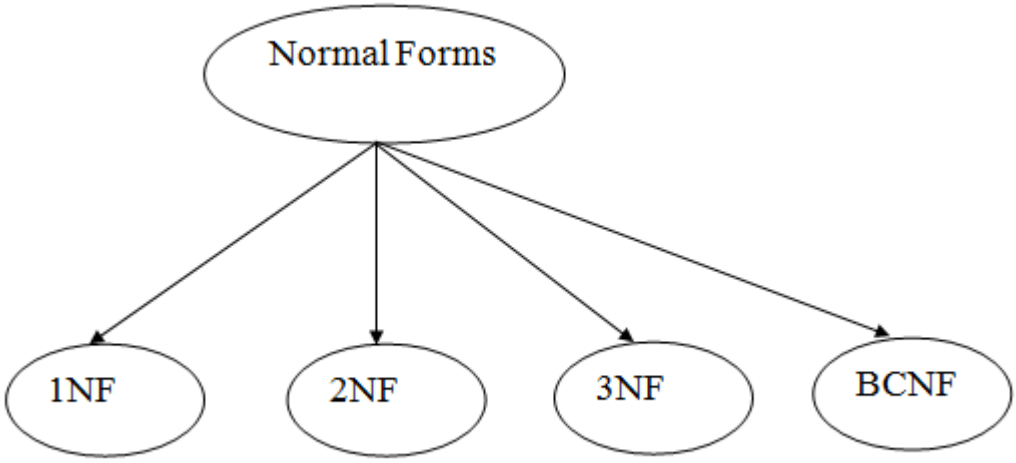
There are the four types of normal forms:

First Normal Form (1NF)

Second Normal Form (2NF)

Third Normal Form (3NF)

Bayce-Codd's Normal Form (BCNF)



Normal Form	Description
1NF	A relation is in 1NF if it contains an atomic (single) value.
2NF	• The table should be in the 1NF. • There should be no Partial Dependency.
3NF	• The table should be in the 2NF. • There should be no transitive Dependency.
BCNF (3.5NF)	• It should be in the 3NF. • And, for any dependency A → B, A should be a super key.

First Normal Form

- 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	Mohan	7272826385, 9064738238	UP
20	Harish	8574783832	Bihar
12	Shyam	7390372389, 8589830302	Punjab

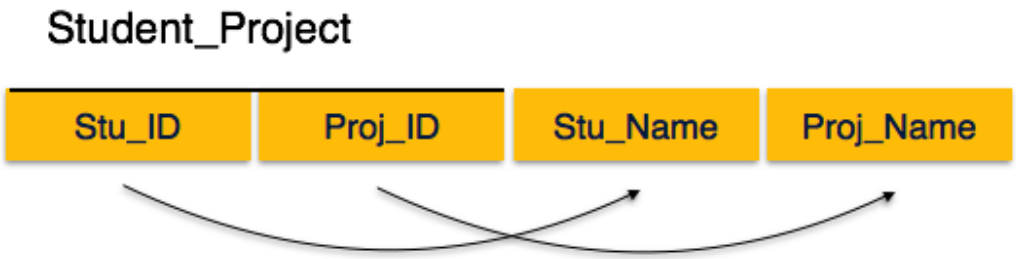
The decomposition of the EMPLOYEE table into 1NF has been shown below:

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	Mohan	7272826385	UP
14	Mohan	9064738238	UP
20	Harish	8574783832	Bihar
12	Shyam	7390372389	Punjab
12	Shyam	8589830302	Punjab

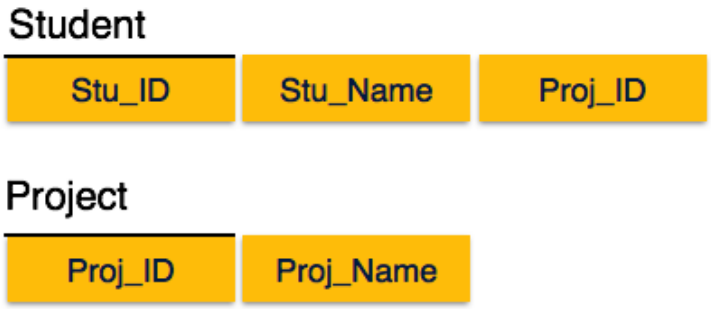
Second Normal Form (2NF)

- In the 2NF, relational must be in 1NF.
- In the second normal form, there should be no Partial Dependency. All non-key attributes should be fully functional dependent on the primary key

If we follow second normal form, then every non-prime attribute should be fully functionally dependent on prime key attribute. That is, if X → A holds, then there should not be any proper subset Y of X, for which Y → A also holds true.



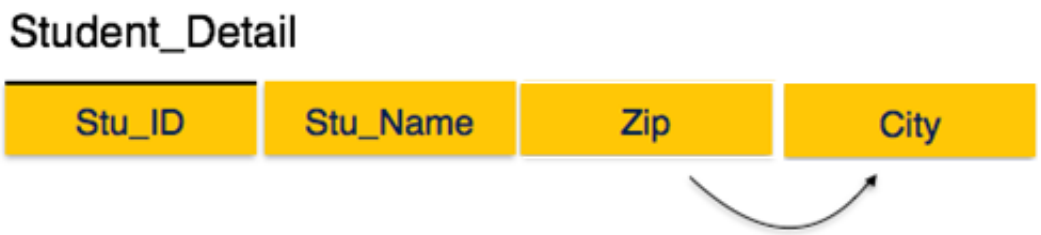
We see here in Student_Project relation that the prime key attributes are Stu_ID and Proj_ID. According to the rule, non-key attributes, i.e. Stu_Name and Proj_Name must be dependent upon both and not on any of the prime key attribute individually. But we find that Stu_Name can be identified by Stu_ID and Proj_Name can be identified by Proj_ID independently. This is called partial dependency, which is not allowed in Second Normal Form.



We broke the relation in two as depicted in the above picture. So there exists no partial dependency.

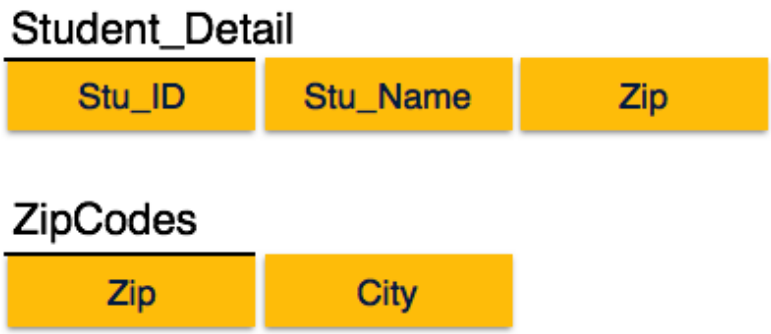
Third Normal Form (3NF)

- 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.



We find that in the above Student_detail relation, Stu_ID is the key and only prime key attribute. We find that Zip can be identified by Stu_ID and as well as city can be identified by Zip itself. Neither Zip is a superkey nor is City a prime attribute. Additionally, Stu_ID → Zip → City, so there exists transitive dependency.

To bring this relation into third normal form, we break the relation into two relations as follows-



Boyce Codd normal form (BCNF) (3.5NF)

BCNF stands for Boyce-Codd normal form and was made by R.F Boyce and E.F Codd in 1947. BCNF is the advance version of 3NF. It is stricter than 3NF.A functional dependency is said to be in BCNF if these properties hold:

- It should already be in 3NF.
- For a functional dependency say P→ Q, P should be a super key.

Example:

Below we have a college enrolment table with columns student_id, subject and professor.

As we can see, we have also added some sample data to the table.

student_id	student_name	Subject	Professor
101	Ram	Java	P.Java1
101	Ram	C++	P.Cpp
102	Mohan	Java	P.Java2
103	Suresh	C#	P.Chash
104	Hitesh	Java	P.Java

In the table above:

- One student can enrol for multiple subjects. For example, student with **student_id** 101 **student_name** Ram, has opted for subjects - Java & C++
- For each subject, a professor is assigned to the student.
- And, there can be multiple professors teaching one subject like we have for Java.

One more important point to note here is, one professor teaches only one subject, but one subject may have two different professors.

Hence, there is a dependency between subject and professor here, where subject depends on the professor name.

This table satisfies the **1st Normal form** because all the values are atomic, column names are unique and all the values stored in a particular column are of same domain.

This table also satisfies the **2nd Normal Form** as there is no **Partial Dependency**.

And, there is no **Transitive Dependency**, hence the table also satisfies the **3rd Normal Form**.

But this table is not in **Boyce-Codd Normal Form**.

Why this table is not in BCNF?

In the table above, student_id, subject are keys, or we can say that student_id and subject are **prime attribute** in this table.

But, there is one more dependency, professor → subject.

And while subject is a **prime attribute**, professor is a **non-prime attribute**, which is not allowed by BCNF.

How to satisfy BCNF?

To make this relation(table) satisfy BCNF, we will decompose this table into two tables, **student** table and **professor** table.

Below we have the structure for both the tables.

Student Table

student_id	**p_id**
101	1
101	2
and so on...	

And, **Professor Table**

p_id	**Professor**	**Subject**
1	P.Java	Java
2	P.Cpp	C++
and so on...		

And now, this relation satisfy Boyce-Codd Normal Form.

P_Id → {professor, subject}

Decomposition

Decomposition in DBMS removes redundancy, anomalies and inconsistencies from a database by dividing the table into multiple tables. Decomposition is the process of breaking down in parts or elements. It replaces a relation with a collection of smaller relations. It breaks the table into multiple tables in a database. It should always be lossless, because it confirms that the information in the original relation can be accurately reconstructed based on the decomposed relations.

- When a relation in the relational model is not in appropriate normal form then the decomposition of a relation is required.

- In a database, it breaks the table into multiple tables.
- If the relation has no proper decomposition, then it may lead to problems like loss of information.
- Decomposition is used to eliminate some of the problems of bad design like anomalies, inconsistencies, and redundancy.

The following are the types of Decomposition −

1. Lossless Decomposition
2. Lossy Decomposition

1. Lossless Decomposition

- If the information is not lost from the relation that is decomposed, then the decomposition will be lossless.
- The lossless decomposition guarantees that the join of relations will result in the same relation as it was decomposed.
- The relation is said to be lossless decomposition if natural joins of all the decomposition give the original relation.

Let us see an example −

<EmpInfo>

Emp_ID	Emp_Name	Emp_Age	Emp_Location	Dept_ID	Dept_Name
E001	Jacob	29	Alabama	Dpt1	Operations
E002	Henry	32	Alabama	Dpt2	HR
E003	Tom	22	Texas	Dpt3	Finance

Decompose the above table into two tables:

<EmpDetails>

Emp_ID	Emp_Name	Emp_Age	Emp_Location
E001	Jacob	29	Alabama
E002	Henry	32	Alabama
E003	Tom	22	Texas

<DeptDetails>

Dept_ID	Emp_ID	Dept_Name
Dpt1	E001	Operations
Dpt2	E002	HR
Dpt3	E003	Finance

Now, Natural Join is applied on the above two tables −

The result will be −

Emp_ID	Emp_Name	Emp_Age	Emp_Location	Dept_ID	Dept_Name
E001	Jacob	29	Alabama	Dpt1	Operations
E002	Henry	32	Alabama	Dpt2	HR
E003	Tom	22	Texas	Dpt3	Finance

Therefore, the above relation had lossless decomposition i.e. no loss of information.

2. Lossy Decomposition

- If the information is lost from the relation that is decomposed, then the decomposition will be lossy.
- The lossy decomposition did not guarantees that the join of relations will result in the same relation as it was decomposed.
- The relation is said to be lossy decomposition if natural joins of all the decomposition did not give the original relation.

"The decomposition of relation R into R1 and R2 is **lossy** when the join of R1 and R2 does not yield the same relation as in R."

One of the disadvantages of decomposition into two or more relational schemes (or tables) is that some information is lost during retrieval of original relation or table.

Let us see an example −

<EmpInfo>

Emp_ID	Emp_Name	Emp_Age	Emp_Location	Dept_ID	Dept_Name
E001	Jacob	29	Alabama	Dpt1	Operations
E002	Henry	32	Alabama	Dpt2	HR
E003	Tom	22	Texas	Dpt3	Finance

Decompose the above table into two tables −

<EmpDetails>

Emp_ID	**Emp_Name**	**Emp_Age**	**Emp_Location**
E001	Jacob	29	Alabama
E002	Henry	32	Alabama
E003	Tom	22	Texas

<DeptDetails>

Dept_ID	**Dept_Name**
Dpt1	Operations
Dpt2	HR
Dpt3	Finance

Now, you won't be able to join the above tables, since **Emp_ID** isn't part of the **DeptDetails** relation.

Therefore, the above relation has lossy decomposition.

Inclusion Dependency in DBMS

Inclusion dependencies (IND) support an essential semantics of the standard relational data model. It is a statement in which a few of the columns of a relation are in the other columns. Typically, inclusion dependency has a very insignificant influence on the design of a database. For example, a foreign key is inclusion dependency. Inclusion Dependency (IND). IND is the rule among different schemas.

Multivalued dependency and join dependency can be used to guide database design although they both are less common than functional dependencies.

Inclusion dependencies are quite common. They typically show little influence on designing of the database.

Transaction Processing In DBMS

Transaction

A transaction can be defined as a group of operations to perform a logical unit of work. A single task is the minimum processing unit which cannot be divided further.

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

For example, we are transferring money from our bank account to our friend's account; the set of operations would be like this:

Simple Transaction Example

1. Read our account balance
2. Deduct the amount from our balance
3. Write the remaining balance to our account
4. Read our friend's account balance
5. Add the amount to his account balance
6. Write the new updated balance to his account

This whole set of operations can be called a transaction.

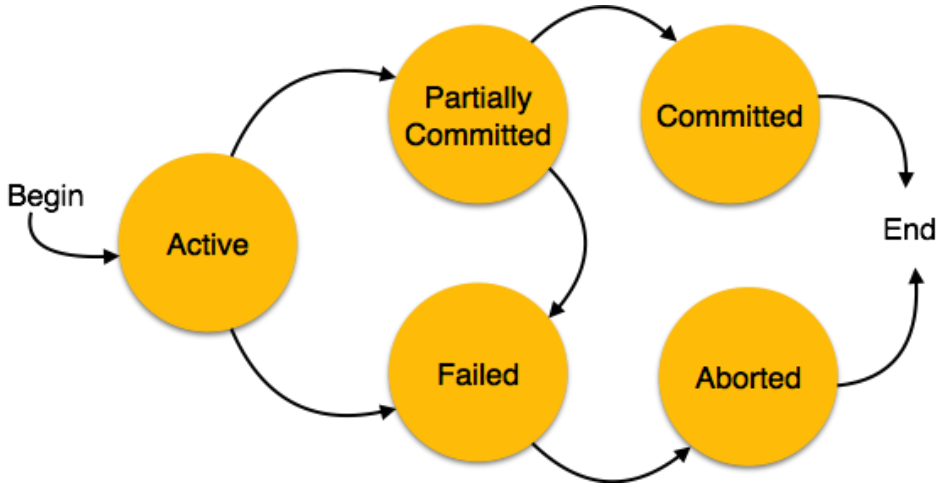
In DBMS, we write the above 6 steps transaction like this:

Lets say your account is A and your friend's account is B, you are transferring 10000 from A to B, the steps of the transaction are:

```
1. R(A);
2. A = A - 10000;
3. W(A);
4. R(B);
5. B = B + 10000;
6. W(B);
```

States of Transactions

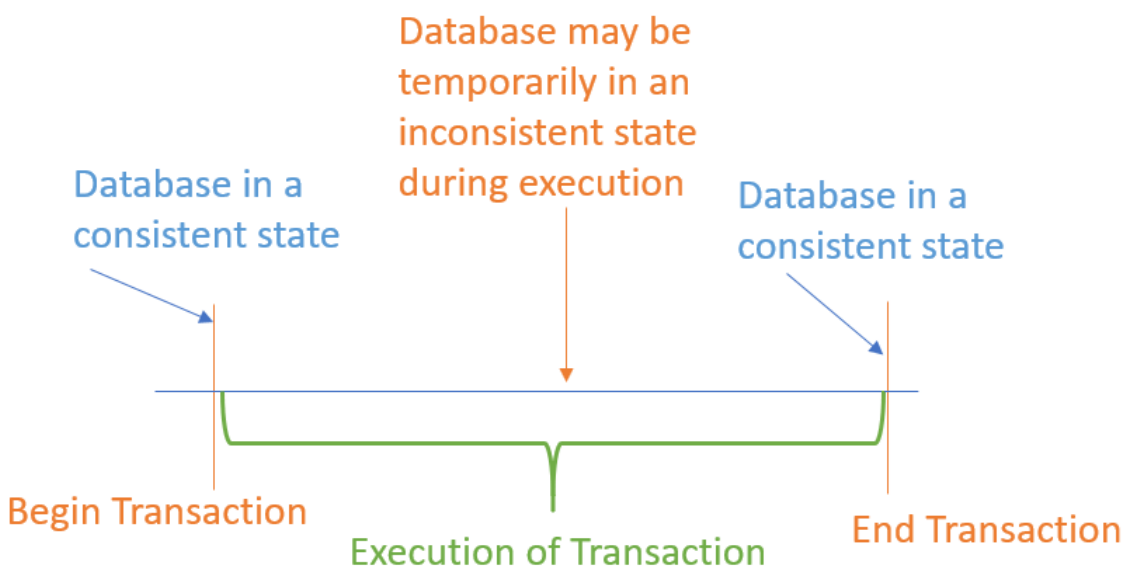
A transaction in a database can be in one of the following states −



- **Active** − In this state, the transaction is being executed. This is the initial state of every transaction.
- **Partially Committed** − When a transaction executes its final operation, it is said to be in a partially committed state.
- **Failed** − A transaction is said to be in a failed state if any of the checks made by the database recovery system fails. A failed transaction can no longer proceed further.
- **Aborted** − If any of the checks fails and the transaction has reached a failed state, then the recovery manager rolls back all its write operations on the database to bring the database back to its original state where it was prior to the execution of the transaction. Transactions in this state are called aborted. The database recovery module can select one of the two operations after a transaction aborts −
    - Re-start the transaction
    - Kill the transaction
- **Committed** − If a transaction executes all its operations successfully, it is said to be committed. All its effects are now permanently established on the database system.

Transaction failure in between the operations

The main problem that can happen during a transaction is that the transaction can fail before finishing the all the operations in the set. This can happen due to power failure, system crash etc. This is a serious problem that can leave database in an inconsistent state.



Assume that transaction fail after third operation (see the example above) then the amount would be deducted from our account but our friend will not receive it.

To solve this problem, we have the following two operations

Commit: If all the operations in a transaction are completed successfully then commit those changes to the database permanently.

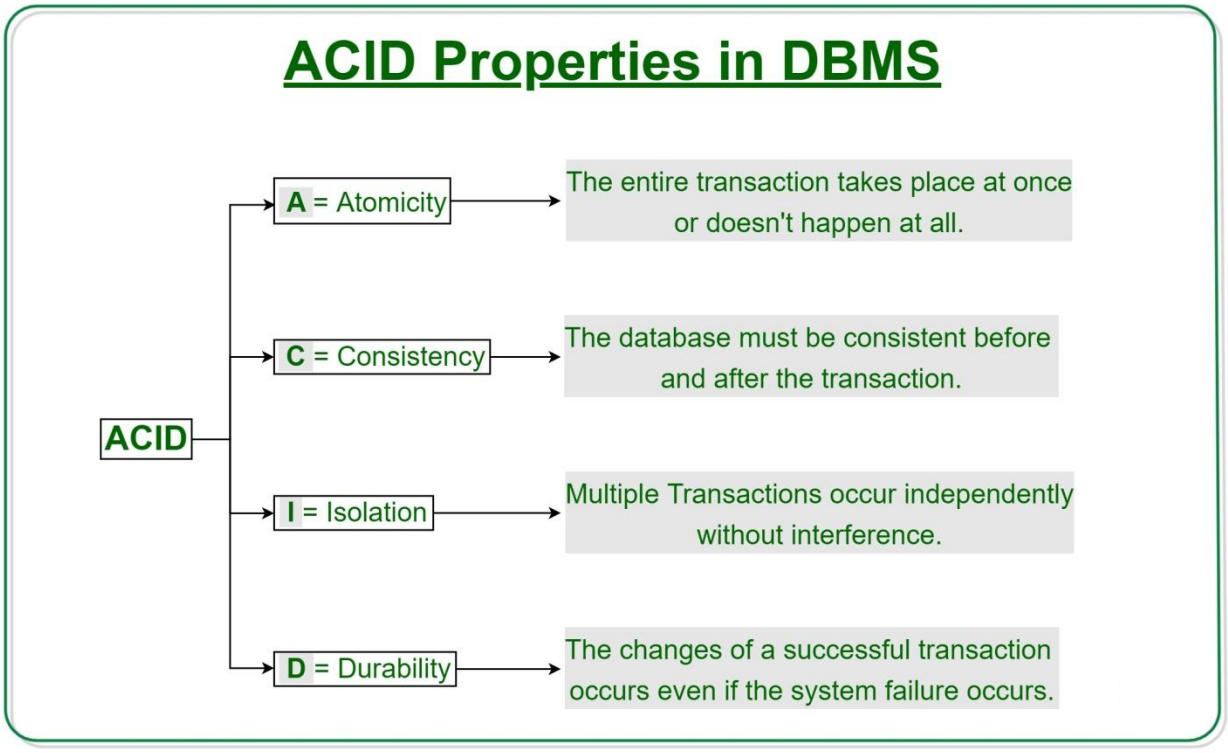
Rollback: If any of the operation fails then rollback all the changes done by previous operations.

Even though these operations can help us avoiding several issues that may arise during transaction but they are not sufficient when two transactions are running concurrently. To handle those problems we need to understand database **ACID properties.**

ACID Properties.

A transaction is a very small unit of a program and it may contain several lowlevel tasks. A transaction in a database system must maintain **A**tomicity, **C**onsistency, **I**solation, and **D**urability − commonly known as ACID properties − in order to ensure accuracy, completeness, and data integrity.

ACID (atomicity, consistency, isolation, durability) is a set of properties of database transactions intended to guarantee data validity despite errors, power failures, and other mishaps.



- **Atomicity** −This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. We either execute it entirely or do not execute it at all. There cannot be partial execution.
  - This property ensures that either the transaction occurs completely or it does not occur at all.
  - In other words, it ensures that no transaction occurs partially.
  - That is why, it is also referred to as "**All or nothing rule**".
  - It is the responsibility of Transaction Control Manager to ensure atomicity of the transactions.

- **Consistency** −The database must remain in a consistent state after any transaction. No transaction should have any undesirable effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.
  - This property ensures that integrity constraints are maintained.
  - In other words, it ensures that the database remains consistent before and after the transaction.

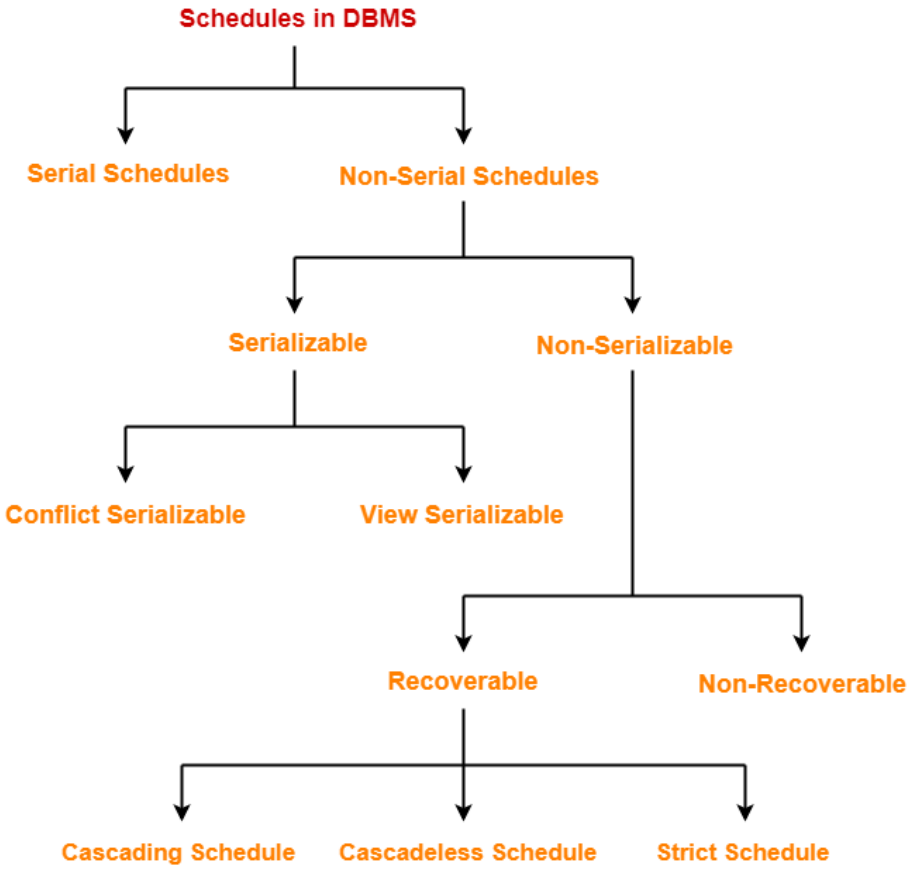
- It is the responsibility of DBMS and application programmer to ensure consistency of the database.

- **Isolation** − In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.
- This property ensures that multiple transactions can occur simultaneously without causing any inconsistency.
- During execution, each transaction feels as if it is getting executed alone in the system.
- A transaction does not realize that there are other transactions as well getting executed parallely.
- Changes made by a transaction becomes visible to other transactions only after they are written in the memory.
- It is the responsibility of concurrency control manager to ensure isolation for all the transactions.

- **Durability** −The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.
- This property ensures that all the changes made by a transaction after its successful execution has written successfully to the disk.
- It also ensures that these changes exist permanently and are never lost even if there occurs a failure of any kind.
- It is the responsibility of recovery manager to ensure durability in the database.

Schedules in DBMS

Schedule − A schedule is the order in which the operations of multiple transactions appear for execution. A schedule can have many transactions in it, each comprising of a number of instructions/tasks. There are two types of schedule in DBMS as:-



1. **Serial Schedule** −In serial schedules, all the transactions execute serially one after the other. When one transaction executes, no other transaction is allowed to execute. Transactions are ordered one after the other. This type of schedule is called a serial schedule, as transactions are executed in a serial manner. Serial schedules are always consistent.

Ex1:

Transaction T1	Transaction T2
R (A)	
W (A)	
R (B)	
W (B)	
Commit	
	R (A)
	W (B)
	Commit

In this schedule,

- There are two transactions T1 and T2 executing serially one after the other.
- Transaction T1 executes first.
- After T1 completes its execution, transaction T2 executes.
- So, this schedule is an example of a **Serial Schedule**.

Ex2:

Transaction T1	Transaction T2
	R (A)
	W (B)
	Commit
R (A)	
W (A)	
R (B)	
W (B)	
Commit	

In this schedule,

- There are two transactions T1 and T2 executing serially one after the other.
- Transaction T2 executes first.
- After T2 completes its execution, transaction T1 executes.
- So, this schedule is an example of a **Serial Schedule.**

2. **Non-serial schedules** - In non-serial schedules, multiple transactions execute concurrently. Operations of all the transactions are interleaved or mixed with each other. Non-serial schedules are not always consistent.

   Ex1:

Transaction T1	Transaction T2
R (A)	
W (B)	
	R (A)
R (B)	
W (B)	
Commit	
	R (B)
	Commit

In this schedule,

- There are two transactions T1 and T2 executing concurrently.
- The operations of T1 and T2 are interleaved.
- So, this schedule is an example of a **Non-Serial Schedule**.

Serializability in DBMS-

When multiple transactions are running concurrently (non-serial Schedule) then there is a possibility that either the database may be left in an inconsistent state or instructions of one transaction are interleaved with some other transaction. Some non-serial schedules may lead to inconsistency of the database.

Serializability is a concept that helps us to check which schedules are serializable. A serializable schedule is the one that always leaves the database in consistent state.

It helps to identify which non-serial schedules are correct and will maintain the consistency of the database.

Serializability of schedules

A serial schedule is always a serializable schedule because any transaction only starts its execution when another transaction has already completed its execution. However, a non-serial schedule of transactions needs to be checked for Serializability.

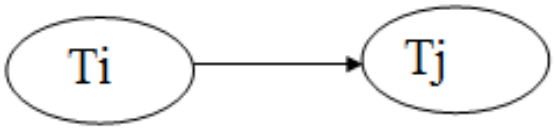
If a schedule of concurrent 'n' transactions can be converted into an equivalent serial schedule. Then we can say that the schedule is serializable. And this property is known as serializability.

Testing for Serializability

To test the serializability of a schedule, we can use the serialization graph. Consider a schedule S. We construct a direct graph, called a precedence graph, from S. This graph consists of a pair G = (V, E), where V is a set of vertices and E is a set of edges. The set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges Ti -> Tj for which one of three conditions holds:

1. Ti executes write (Q) before Tj executes read (Q). **(Write(Q) - Read(Q))**
2. Ti executes read (Q) before Tj executes write (Q). **(Read(Q) -Write(Q))**
3. Ti executes write (Q) before Tj executes write (Q). **(Write(Q) -Write(Q))**

Precedence graph for Schedule S

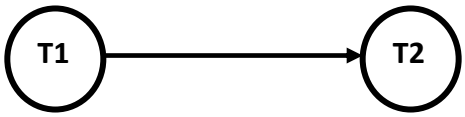


Schedule S Precedence Graph

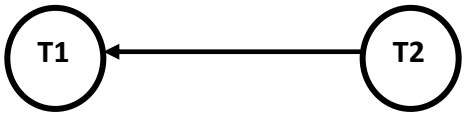
- If a precedence graph contains a single edge Ti → Tj, then all the instructions of Ti are executed before the first instruction of Tj is executed.
- If a precedence graph for schedule S contains a cycle, then S is non-serializable. If the precedence graph has no cycle, then S is known as serializable.

Example1: Serialization graphs (precedence graph) for serial schedules S having two transactions T1 & T2-

Transaction T1	Transaction T2
R (A)	
W (A)	
R (B)	
W (B)	
Commit	
	R (A)
	W (B)
	Commit

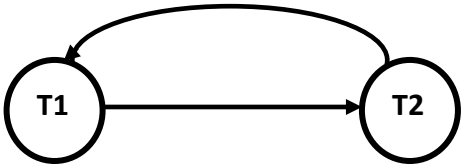


Transaction T1	Transaction T2
	R (A)
	W (B)
	Commit
R (A)	
W (A)	
R (B)	
W (B)	
Commit	

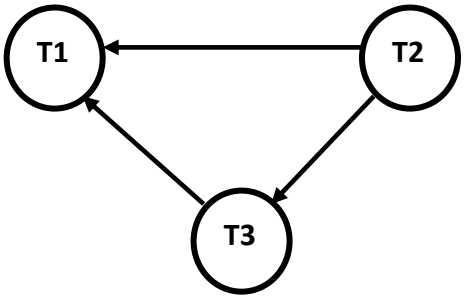


Example2: Serialization graphs (precedence graphs) for non-serial schedule S having two transactions T1 & T2-

Transaction T1	Transaction T2
R(A)	
	R(A)
	W(A)
W(A)	

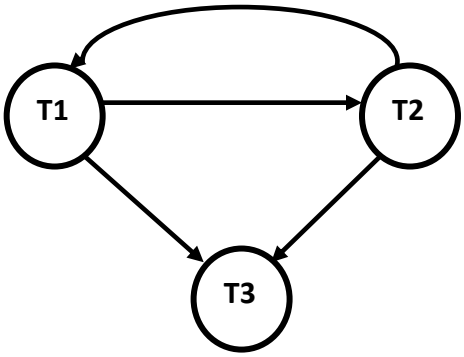


Transaction T1	Transaction T2	Transaction T3
R(A)		
		R(B)
		R(A)
	R(B)	
	R(C)	
		W(B)
	W(C)	
R(C)		
W(A)		
W(C)		



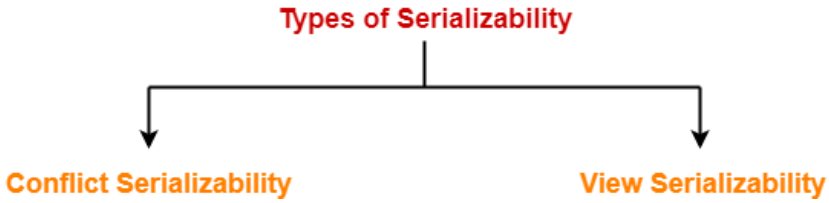
Example2: Serialization graphs (precedence graphs) for a non-serial schedule S having three transactions T1, T2 & T3-

Transaction T1	Transaction T2	Transaction T3
R(A)		
	W(A)	
W(A)		
		W(A)



Types of Serializability-

Serializability is mainly of two types-



1. Conflict Serializability
2. View Serializability

Conflict Serializability-

If a given non-serial schedule can be converted into a serial schedule by swapping its non-conflicting operations, then it is called as a **conflict serializable schedule**.

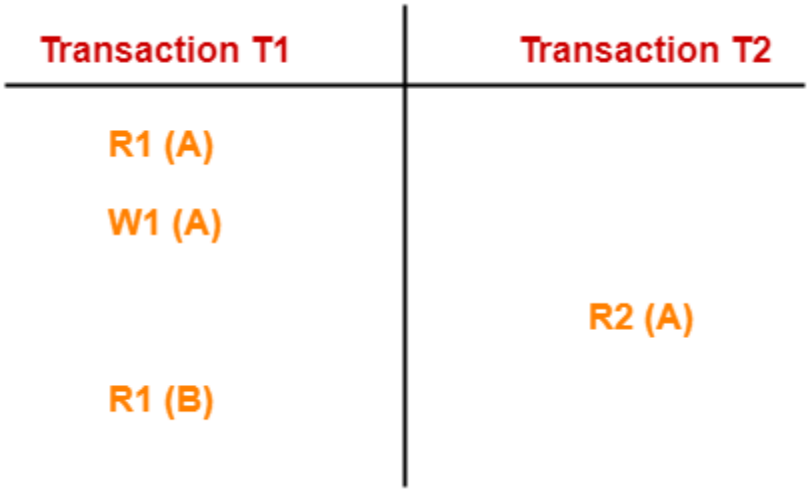
Conflicting Operations-

Two operations are called as **conflicting operations** if all the following conditions hold true for them-

- Both the operations belong to different transactions
- Both the operations are on the same data item
- At least one of the two operations is a write operation

Example-

Consider the following schedule-



In this schedule,

- W1 (A) and R2 (A) are called as conflicting operations.
- This is because all the above conditions hold true for them.

Example

Non-serial schedule

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

Schedule S1

Serial Schedule

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

Schedule S2

Schedule S2 is a serial schedule because, in this, all operations of T1 are performed before starting any operation of T2. Schedule S1 can be transformed into a serial schedule by swapping non-conflicting operations of S1.

After swapping of non-conflict operations, the schedule S1 becomes:

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

Since, S1 is conflict serializable.

Example

Let us consider the following schedule and see if it is serializable.

Transaction T1	Transaction T2
	R(X)
R(X)	
	R(Y)
	W(Y)
R(Y)	
W(X)	

Now, let us figure out if the above schedule is serializable.

1. Swapping R(X) of T1 and R(Y) of T2.
2. Swapping R(X) of T1 and W(Y) of T2.

Transaction T1	Transaction T2
	R(X)
	R(Y)
	W(Y)
R(X)	
R(Y)	
W(X)	

Thus, after changing the conflicting operations in the initial schedule we get a serial schedule. Hence, this schedule is serializable.

Another Example for Conflict Serializability –

Let us consider the following transaction schedule and test it for Conflict Serializability.

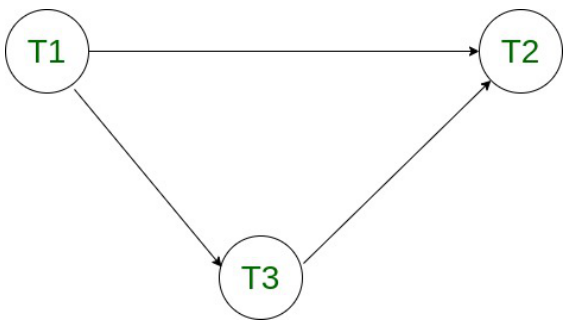
Transaction T1	Transaction T2	Transaction T3
	R(X)	
		R(X)
W(Y)		
	W(X)	
		R(Y)
	W(Y)	

Now, we will list all the conflicting operations. Further, we will determine whether the schedule is conflict serializable using Precedence Graph.

Two operations are said to be conflicting if the belong to different transaction, operate on same data and at least one of them is a read operation.

1. R3(X) and W2(X) [T3 -> T2]
2. W1(Y) and R3(Y) [T1 -> T3]
3. W1(Y) and W2(Y) [T1 -> T2]
4. R3(Y) and W2(Y) [T3 -> T2]

Constructing the precedence graph, we see there are no cycles in the graph. Therefore, the schedule is Conflict Serializable.



View Serializability

View Serializability is a process to find out that a given schedule is view serializable or not.

To check whether a given schedule is view serializable, we need to check whether the given schedule is **View Equivalent** to its serial schedule. If a schedule is conflict serializable, then it is surely view serializable. If a given schedule S is not conflict serializable, it may or may not be view serializable.

View Equivalent

Lets learn how to check whether the two schedules are view equivalent.

Two schedules S1 and S2 are said to be view equivalent, if they satisfy all the following conditions:

1. Initial Read: Initial read of each data item in transactions must match in both schedules. For example, if transaction T1 reads a data item X before transaction T2 in schedule S1 then in schedule S2, T1 should read X before T2.

Read vs Initial Read: Here initial read means the first read operation on a data item, for example, a data item X can be read multiple times in a schedule but the first read operation on X is called the initial read. This will be more clear once we will get to the example in the next section of this same article.

Thumb Rule

"Initial readers must be same for all the data items".

2. Final Write: Final write operations on each data item must match in both the schedules. For example, a data item X is last written by Transaction T1 in schedule S1 then in S2, the last write operation on X should be performed by the transaction T1.

Thumb Rule

"Final writers must be same for all the data items".

3. Update Read: If in schedule S1, the transaction T1 is reading a data item updated by T2 then in schedule S2, T1 should read the value after the write operation of T2 on same data item. For example, In schedule S1, T1 performs a read operation on X after the write operation on X by T2 then in S2, T1 should read the X after T2 performs write on X.

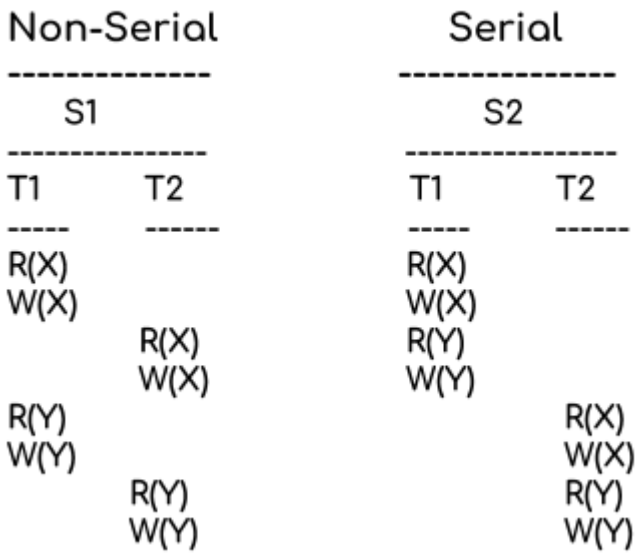
Thumb Rule

"Write-read sequence must be same."

View Serializable

If a schedule is view equivalent to its serial schedule then the given schedule is said to be View Serializable. Lets take an example.

View Serializable Example



Lets check the three conditions of view serializability:

Initial Read

In schedule S1, transaction T1 first reads the data item X. In S2 also transaction T1 first reads the data item X.

Lets check for Y. In schedule S1, transaction T1 first reads the data item Y. In S2 also the first read operation on Y is performed by T1.

We checked for both data items X & Y and the **initial read** condition is satisfied in S1 & S2.

Final Write

In schedule S1, the final write operation on X is done by transaction T2. In S2 also transaction T2 performs the final write on X.

Lets check for Y. In schedule S1, the final write operation on Y is done by transaction T2. In schedule S2, final write on Y is done by T2.

We checked for both data items X & Y and the **final write** condition is satisfied in S1 & S2.

Update Read

In S1, transaction T2 reads the value of X, written by T1. In S2, the same transaction T2 reads the X after it is written by T1.

In S1, transaction T2 reads the value of Y, written by T1. In S2, the same transaction T2 reads the value of Y after it is updated by T1.

The update read condition is also satisfied for both the schedules.

Result: Since all the three conditions that checks whether the two schedules are view equivalent are satisfied in this example, which means S1 and S2 are view equivalent. Also, as we know that the schedule S2 is the serial schedule of S1, thus we can say that the schedule S1 is view serializable schedule.

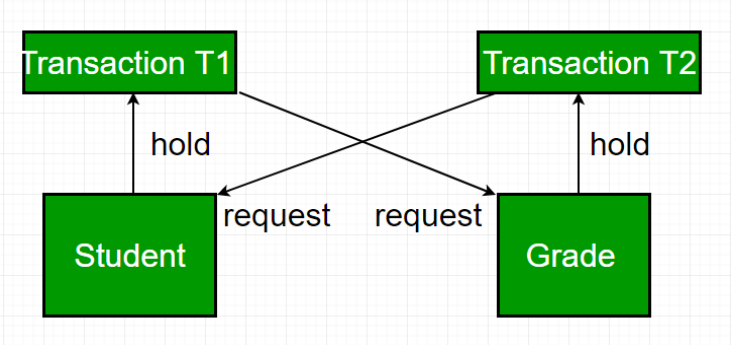
Transaction failures

The transaction failure occurs when it fails to execute some of its operations or when it reaches a point from where it can't go any further. Sometimes a transaction may not execute completely due to a software issue, system crash or hardware failure. In that case, the failed transaction has to be rollback.

If a few transaction or process is hurt, then this is called as transaction failure.

Reasons for a transaction failure could be -

1. **Logical errors:** If a transaction cannot complete due to some code error or an internal error condition, then the logical error occurs. For example any value divided by zero (n/0), or any integer value having range from 1 to 100, and if we put value out of range like -50,105 then due to such type of logical error transaction fails.
2. **Concurrency Control Enforcement:** It occurs where the DBMS itself terminates an active transaction because the database system is not able to execute it. **For example,** The system aborts an active transaction, in case of deadlock or resource unavailability.



3. **System Failure/Crash -**System failure can occur due to many numbers of reasons as power failure, network failure or other hardware or software failure.
4. **Local Errors and Exception-** Local errors occurs when a transaction begins and it between systems realize an exception. For example- we are going to withdraw Rs.5000 from our account and we are having just Rs 4000 in our account, then transaction will fail due to local error or exception error.
5. **Hard-Disk Failure-** Hard drives are generally used to store large files, back up data, and to save important data to keep them safe and secure. A hard disk drive failure occurs when a hard disk drive malfunctions and the stored information cannot be accessed with a properly configured computer. There are a number of causes for hard drives to fail including: manufacturing error, heat, water damage, power issues and mishaps. Other reasons of disk failure occur due to the formation of bad sectors, disk head crash, and unreachability to the disk.
6. **Physical problem or catastrophe-** If whole system destroys due to some external issues like natural disaster as earth-quake or flood then transaction will fail.

Recovery from transaction failures

Here we will discuss two method of transaction recovery

1. **Log-Based Recovery**
    - The log is a sequence of records. Log of each transaction is maintained in some stable storage so that if any failure occurs, then it can be recovered from there.
    - If any operation is performed on the database, then it will be recorded in the log.
    - But the process of storing the logs should be done before the actual transaction is applied in the database.
    - And logs may or may not contain the record of reading of data item. It is so because reading the data item does not affect the consistency of the database and is nowhere helpful in recovery mechanism.

An update log record represented as: **<Ti, Xj, V1, V2>** has these fields:

1. **Transaction identifier:** Unique Identifier of the transaction that performed the write operation.
2. **Data item:** Unique identifier of the data item written.

3. **Old value:** Value of data item prior to write.
4. **New value:** Value of data item after write operation.

Other type of log records is:

<Ti start>	**:**	It contains information about when a transaction Ti starts.
<Ti commit>	**:**	It contains information about when a transaction Ti commits.
<Ti abort>	**:**	It contains information about when a transaction Ti aborts

Undo and Redo Operations –

Because all database modifications must be preceded by creation of log record, the system has available both the old value prior to modification of data item and new value that is to be written for data item. This allows system to perform redo and undo operations as appropriate:

1. **Undo:** using a log record sets the data item specified in log record to old value.
2. **Redo:** using a log record sets the data item specified in log record to new value.

After a system crash has occurred, the system consults the log to determine which transactions need to be redone and which need to be undone.

1. Transaction Ti needs to be undone if the log contains the record <Ti start> but does not contain either the record <Ti commit> or the record <Ti abort>.
2. Transaction Ti needs to be redone if log contains record <Ti start> and either the record <Ti commit> or the record <Ti abort>.

Example

```
<To start>
<To A, 1000, 950>
<To B, 2000, 2050>
<To commit>
<T1 start>
<T1 C, 700, 600>
```

We consider an example of banking system taken earlier for transaction To and T1 such that To is followed by T1. If the system crash occurs just after the log record and during recovery we do redo (To) and undo (T1) as we have both < To start > and <To commit> in the log record. But we do not have <T1 commit> with <T1 start> in log record. Undo (T1) should be done first then redo (To) should be done.

2. Checkpoints-Based Recovery

When a system crash occurs, user must consult the log. In principle, that needs to search the entire log to determine this information. There are two major difficulties with this approach:

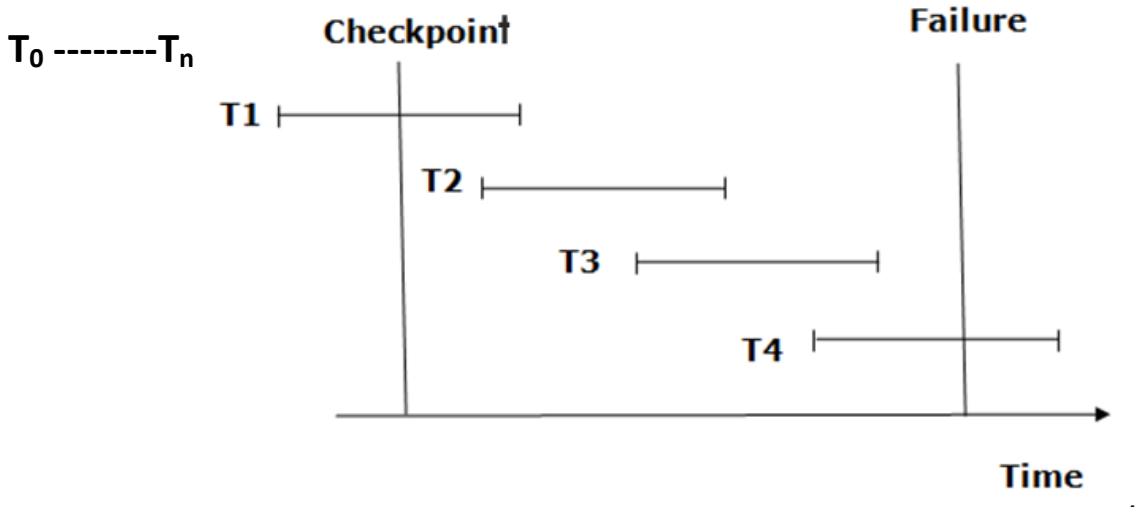
1. The search process is time-consuming.
2. Most of the transactions that need to be redone have already written their updates into the database. Although redoing them will cause no harm, it will cause recovery to take longer.

To reduce these types of overhead, user introduce checkpoints.

- The checkpoint is a type of mechanism where all the previous logs are removed from the system and permanently stored in the storage disk.
- The checkpoint is like a bookmark. While the execution of the transaction, such checkpoints are marked.

- When it reaches to the checkpoint, then the transaction will be updated into the database, and till that point, the entire log file will be removed from the file. Then the log file is updated with the new step of transaction till next checkpoint and so on.
- The checkpoint is used to declare a point before which the DBMS was in the consistent state, and all transactions were committed.

In the following manner, a recovery system recovers the database from this failure:



- The recovery system reads log files from the end to start. It reads log files from T4 to T1.
- Recovery system maintains two lists, a redo-list, and an undo-list.
- The transaction is put into redo state if the recovery system sees a log with <Tn, Start> and <Tn, Commit> or just <Tn, Commit>. In the redo-list and their previous list, all the transactions are removed and then redone before saving their logs.
- **For example:** In the log file, transaction T2 and T3 will have <Tn, Start> and <Tn, Commit>. The T1 transaction will have only <Tn, commit> in the log file. That's why the transaction is committed after the checkpoint is crossed. Hence it puts T1, T2 and T3 transaction into redo list.
- The transaction is put into undo state if the recovery system sees a log with <Tn, Start> but no commit or abort log found. In the undo-list, all the transactions are undone, and their logs are removed.
- **For example:** Transaction T4 will have <Tn, Start>. So T4 will be put into undo list since this transaction is not yet complete and failed in the middle of it.