UNIT-3

Types of Functional dependencies in DBMS

Functional dependency and attribute closure

In a relational database management, functional dependency is a concept that specifies the relationship between two sets of attributes where one attribute determines the value of another attribute. It is denoted as **X → Y**, where the attribute set on the left side of the arrow, **X** is called **Determinant**, and **Y** is called the **Dependent**.

Functional dependencies are used to mathematically express relations among database entities and are very important to understand advanced concepts in Relational Database System and understanding problems in competitive exams like Gate.

Example:

roll_no	name	dept_name	dept_building
42	abc	CO	A4
43	pqr	IT	A3
44	xyz	CO	A4
45	xyz	IT	A3
46	mno	EC	B2
47	jkl	ME	B2

From the above table we can conclude some valid functional dependencies:

- roll_no → { name, dept_name, dept_building },→ Here, roll_no can determine values of fields name, dept_name and dept_building, hence a valid Functional dependency
- roll_no → dept_name , Since, roll_no can determine whole set of {name, dept_name, dept_building}, it can determine its subset dept_name also.
- dept_name → dept_building , Dept_name can identify the dept_building accurately, since departments with different dept_name will also have a different dept_building
- More valid functional dependencies: roll_no → name, {roll_no, name} ⇢ {dept_name, dept_building}, etc.

Here are some invalid functional dependencies:

- name → dept_name Students with the same name can have different dept_name, hence this is not a valid functional dependency.
- dept_building → dept_name There can be multiple departments in the same building. Example, in the above table departments ME and EC are in the same building B2, hence dept_building → dept_name is an invalid functional dependency.
- More invalid functional dependencies: name → roll_no, {name, dept_name} → roll_no, dept_building → roll_no, etc.

Armstrong's axioms/properties of functional dependencies:

1. **Reflexivity:** If Y is a subset of X, then X→Y holds by reflexivity rule Example, {roll_no, name} → name is valid.
2. **Augmentation:** If X → Y is a valid dependency, then XZ → YZ is also valid by the augmentation rule.
   Example, {roll_no, name} → dept_building is valid, hence {roll_no, name, dept_name} → {dept_building, dept_name} is also valid.
3. **Transitivity**: If X → Y and Y → Z are both valid dependencies, then X→Z is also valid by the Transitivity rule.
   Example, roll_no → dept_name & dept_name → dept_building, then roll_no → dept_building is also valid.

Types of Functional Dependencies in DBMS

1. Trivial functional dependency
2. Non-Trivial functional dependency
3. Multivalued functional dependency
4. Transitive functional dependency

1. Trivial Functional Dependency

In **Trivial Functional Dependency**, a dependent is always a subset of the determinant. i.e. If **X → Y** and **Y is the subset of X**, then it is called trivial functional dependency

Example:

roll_no	name	age
42	abc	17
43	pqr	18
44	xyz	18

Here, **{roll_no, name} → name** is a trivial functional dependency, since the dependent **name** is a subset of determinant set **{roll_no, name}.** Similarly, **roll_no → roll_no** is also an example of trivial functional dependency.

2. Non-trivial Functional Dependency

In **Non-trivial functional dependency**, the dependent is strictly not a subset of the determinant. i.e. If **X → Y** and **Y is not a subset of X**, then it is called Non-trivial functional dependency.

Example:

roll_no	name	age
42	abc	17
43	pqr	18
44	xyz	18

Here, **roll_no → name** is a non-trivial functional dependency, since the dependent **name** is **not a subset of** determinant **roll_no.** Similarly, **{roll_no, name} → age** is also a non-trivial functional dependency, since **age** is **not a subset of {roll_no, name}**

3. Multivalued Functional Dependency

In **Multivalued functional dependency**, entities of the dependent set are **not dependent on each other.** i.e. If **a → {b, c}** and there exists **no functional dependency** between **b and c**, then it is called a **multivalued functional dependency.**

For example,

roll_no	name	age
42	abc	17
43	pqr	18
44	xyz	18
45	abc	19

Here, **roll_no → {name, age}** is a multivalued functional dependency, since the dependents **name** & **age** are **not dependent** on each other(i.e. **name → age** or **age → name doesn't exist !**)

4. Transitive Functional Dependency

In transitive functional dependency, dependent is indirectly dependent on determinant. i.e. If **a → b** & **b → c**, then according to axiom of transitivity, **a → c**. This is a **transitive functional dependency.**

For example,

enrol_no	name	dept	building_no
42	abc	CO	4
43	pqr	EC	2
44	xyz	IT	1
45	abc	EC	2

Here, **enrol_no → dept** and **dept → building_no.** Hence, according to the axiom of transitivity, **enrol_no → building_no** is a valid functional dependency. This is an indirect functional dependency, hence called Transitive functional dependency.

5. Fully Functional Dependency

In full functional dependency an attribute or a set of attributes uniquely determines another attribute or set of attributes. If a relation R has attributes X, Y, Z with the dependencies X->Y and X->Z which states that those dependencies are fully functional.

6. Partial Functional Dependency

In partial functional dependency a non key attribute depends on a part of the composite key, rather than the whole key. If a relation R has attributes X, Y, Z where X and Y are the composite key and Z is non key attribute. Then X->Z is a partial functional dependency in RBDMS.

Advantages of Functional Dependencies

Functional dependencies having numerous applications in the field of database management system. Here are some applications listed below:

1. Data Normalization

Data normalization is the process of organizing data in a database in order to minimize redundancy and increase data integrity. Functional dependencies play an important part in data normalization. With the help of functional dependencies we are able to identify the primary key, candidate key in a table which in turns helps in normalization.

2. Query Optimization

With the help of functional dependencies we are able to decide the connectivity between the tables and the necessary attributes need to be projected to retrieve the required data from the tables. This helps in query optimization and improves performance.

3. Consistency of Data

Functional dependencies ensures the consisncy of ttehe data by removing any redundancies or inconsistencies that may exist in the data. Functional dependency ensures that the changes made in one attribute does not affect inconsistency in another set of attributes thus it maintains the consistency of the data in database.

4. Data Quality Improvement

Functional dependencies ensure that the data in the database to be accurate, complete and updated. This helps to improve the overall quality of the data, as well as it eliminates errors and inaccuracies that might occur during data analysis and decision making, thus functional dependency helps in improving the quality of data in database

Introduction of Database Normalization

Database normalization is the process of organizing the attributes of the database to reduce or eliminate **data redundancy (having the same data but at different places)**.
Problems because of data redundancy: Data redundancy unnecessarily increases the size of the database as the same data is repeated in many places. Inconsistency problems also arise during insert, delete and update operations.
Functional Dependency: Functional Dependency is a constraint between two sets of attributes in relation to a database. A functional dependency is denoted by an arrow (→). If an attribute A functionally determines B, then it is written as A → B.
For example, employee_id → name means employee_id functionally determines the name of the employee. As another example in a timetable database, {student_id, time} → {lecture_room}, student ID and time determine the lecture room where the student should be.

Advantages of Functional Dependency

- The database's data quality is maintained using it.
- It communicates the database design's facts.
- It aids in precisely outlining the limitations and implications of databases.
- It is useful to recognize poor designs.
- Finding the potential keys in the relationship is the first step in the normalization procedure. Identifying potential keys and normalizing the database without functional dependencies is impossible.

What does functionally dependent mean?
A function dependency A → B means for all instances of a particular value of A, there is the same value of B. For example in the below table A → B is true, but B → A is not true as there are different values of A for B = 3.

A	B

```
------

1   3

2   3

4   0

1   3

4   0
```

Trivial Functional Dependency
X → Y is trivial only when Y is a subset of X.

Examples

```
ABC → AB

ABC → A

ABC → ABC
```

Non Trivial Functional Dependencies
```
X → Y is a non-trivial functional dependency when Y is not a subset of
X.

X → Y is called completely non-trivial when X intersect Y is NULL.
```

Example:
```
Id → Name,

Name → DOB
```

Semi Non Trivial Functional Dependencies
```
X → Y is called semi non-trivial when X intersect Y is not NULL.
```

Examples:

```
AB → BC,

AD → DC
```

- Normal Forms
- Quiz on Normalization

The features of database normalization are as follows:
Elimination of Data Redundancy: One of the main features of normalization is to eliminate the data redundancy that can occur in a database. Data redundancy refers to the repetition of data in different parts of the database. Normalization helps in reducing or eliminating this redundancy, which can improve the efficiency and consistency of the database.

Ensuring Data Consistency: Normalization helps in ensuring that the data in the database is consistent and accurate. By eliminating redundancy, normalization helps in preventing inconsistencies and contradictions that can arise due to different versions of the same data.
Simplification of Data Management: Normalization simplifies the process of managing data in a database. By breaking down a complex data structure into simpler tables, normalization makes it easier to manage the data, update it, and retrieve it.
Improved Database Design: Normalization helps in improving the overall design of the database. By organizing the data in a structured and systematic way, normalization makes it easier to design and maintain the database. It also makes the database more flexible and adaptable to changing business needs.
Avoiding Update Anomalies: Normalization helps in avoiding update anomalies, which can occur when updating a single record in a table affects multiple records in other tables. Normalization ensures that each table contains only one type of data and that the relationships between the tables are clearly defined, which helps in avoiding such anomalies.
Standardization: Normalization helps in standardizing the data in the database. By organizing the data into tables and defining relationships between them, normalization helps in ensuring that the data is stored in a consistent and uniform manner.
Normalization is an important process in database design that helps in improving the efficiency, consistency, and accuracy of the database. It makes it easier to manage and maintain the data and ensures that the database is adaptable to changing business needs.

Normal Forms in DBMS

Normalization is the process of minimizing **redundancy** from a relation or set of relations. Redundancy in relation may cause insertion, deletion, and update anomalies. So, it helps to minimize the redundancy in relations. **Normal forms** are used to eliminate or reduce redundancy in database tables.

Introduction: In database management systems (DBMS), normal forms are a series of guidelines that help to ensure that the design of a database is efficient, organized, and free from data anomalies. There are several levels of normalization, each with its own set of guidelines, known as normal forms.

Here are the important points regarding normal forms in DBMS:

1. First Normal Form (1NF): This is the most basic level of normalization. In 1NF, each table cell should contain only a single value, and each column should have a unique name. The first normal form helps to eliminate duplicate data and simplify queries.
2. Second Normal Form (2NF): 2NF eliminates redundant data by requiring that each non-key attribute be dependent on the primary key. This means that each column should be directly related to the primary key, and not to other columns.

3. Third Normal Form (3NF): 3NF builds on 2NF by requiring that all non-key attributes are independent of each other. This means that each column should be directly related to the primary key, and not to any other columns in the same table.
4. Boyce-Codd Normal Form (BCNF): BCNF is a stricter form of 3NF that ensures that each determinant in a table is a candidate key. In other words, BCNF ensures that each non-key attribute is dependent only on the candidate key.
5. Fourth Normal Form (4NF): 4NF is a further refinement of BCNF that ensures that a table does not contain any multi-valued dependencies.
6. Fifth Normal Form (5NF): 5NF is the highest level of normalization and involves decomposing a table into smaller tables to remove data redundancy and improve data integrity.

Normal forms help to reduce data redundancy, increase data consistency, and improve database performance. However, higher levels of normalization can lead to more complex database designs and queries. It is important to strike a balance between normalization and practicality when designing a database

The advantages of using normal forms in DBMS include:

- Reduced data redundancy: Normalization helps to eliminate duplicate data in tables, reducing the amount of storage space needed and improving database efficiency.
- Improved data consistency: Normalization ensures that data is stored in a consistent and organized manner, reducing the risk of data inconsistencies and errors.
- Simplified database design: Normalization provides guidelines for organizing tables and data relationships, making it easier to design and maintain a database.
- Improved query performance: Normalized tables are typically easier to search and retrieve data from, resulting in faster query performance.
- Easier database maintenance: Normalization reduces the complexity of a database by breaking it down into smaller, more manageable tables, making it easier to add, modify, and delete data.

Overall, using normal forms in DBMS helps to improve data quality, increase database efficiency, and simplify database design and maintenance.

1. First Normal Form –

If a relation contain composite or multi-valued attribute, it violates first normal form or a relation is in first normal form if it does not contain any composite or multi-valued attribute. A relation is in first normal form if every attribute in that relation is **singled valued attribute**.

- **Example 1 –** Relation STUDENT in table 1 is not in 1NF because of multi-valued attribute STUD_PHONE. Its decomposition into 1NF has been shown in table

2.

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD_COUNTRY
1	RAM	9716271721, 9871717178	HARYANA	INDIA
2	RAM	9898297281	PUNJAB	INDIA
3	SURESH		PUNJAB	INDIA

Table 1

Conversion to first normal form

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD_COUNTRY
1	RAM	9716271721	HARYANA	INDIA
1	RAM	9871717178	HARYANA	INDIA
2	RAM	9898297281	PUNJAB	INDIA
3	SURESH		PUNJAB	INDIA

Table 2

- **Example 2 –**

```
ID   Name   Courses
------------------
1    A      c1, c2
2    E      c3
3    M      C2, c3
```

- In the above table Course is a multi-valued attribute so it is not in 1NF. Below Table is in 1NF as there is no multi-valued attribute

```
ID   Name   Course
------------------
1    A       c1
1    A       c2
2    E       c3
3    M       c2
3    M       c3
```

-

2. Second Normal Form –

To be in second normal form, a relation must be in first normal form and relation must not contain any partial dependency. A relation is in 2NF if it has **No Partial Dependency,** i.e.**,** no non-prime attribute (attributes which are not part of any candidate key) is dependent on any proper subset of any candidate key of the table. **Partial Dependency –** If the proper subset of candidate key determines non-prime attribute, it is called partial dependency.

- **Example 1 –** Consider table-3 as following below.

STUD_NO	COURSE_NO	COURSE_FEE
1	C1	1000
2	C2	1500
1	C4	2000
4	C3	1000
4	C1	1000
2	C5	2000

- {Note that, there are many courses having the same course fee. } Here, COURSE_FEE cannot alone decide the value of COURSE_NO or STUD_NO; COURSE_FEE together with STUD_NO cannot decide the value of COURSE_NO; COURSE_FEE together with COURSE_NO cannot decide the value of STUD_NO; Hence, COURSE_FEE would be a non-prime attribute, as it does not belong to the one only candidate key {STUD_NO, COURSE_NO} ; But, COURSE_NO -> COURSE_FEE, i.e., COURSE_FEE is dependent on COURSE_NO, which is a proper subset of the candidate key. Non-prime attribute COURSE_FEE is dependent on a proper subset of the candidate key, which is a partial dependency and so this relation is not in 2NF. To convert the above relation to 2NF, we need to split the table into two tables such as : Table 1: STUD_NO, COURSE_NO Table 2: COURSE_NO, COURSE_FEE

Table 1

STUD_NO	COURSE_NO
1	C1
2	C2
1	C4
4	C3
4	C1
2	C5

Table 2

COURSE_NO	COURSE_FEE
C1	1000
C2	1500
C3	1000
C4	2000
C5	2000

- **NOTE:** 2NF tries to reduce the redundant data getting stored in memory. For instance, if there are 100 students taking C1 course, we don't need to store its Fee as 1000 for all the 100 records, instead, once we can store it in the second table as the course fee for C1 is 1000.
- **Example 2 –** Consider following functional dependencies in relation R (A, B , C, D)

```
AB -> C  [A and B together determine C]

BC -> D  [B and C together determine D]
```

- In the above relation, AB is the only candidate key and there is no partial dependency, i.e., any proper subset of AB doesn't determine any non-prime attribute.
  1. X is a super key.
  2. Y is a prime attribute (each element of Y is part of some candidate key).
- **Example 1 –** In relation STUDENT given in Table 4, FD set: {STUD_NO -> STUD_NAME, STUD_NO -> STUD_STATE, STUD_STATE -> STUD_COUNTRY, STUD_NO -> STUD_AGE} Candidate Key: {STUD_NO} For this relation in table 4, STUD_NO -> STUD_STATE and STUD_STATE -> STUD_COUNTRY are true. So STUD_COUNTRY is transitively dependent on STUD_NO. It violates the third normal form. To convert it in third normal form, we will decompose the relation STUDENT (STUD_NO, STUD_NAME, STUD_PHONE, STUD_STATE, STUD_COUNTRY_STUD_AGE) as: STUDENT (STUD_NO, STUD_NAME, STUD_PHONE, STUD_STATE, STUD_AGE) STATE_COUNTRY (STATE, COUNTRY)
- **Example 2 –** Consider relation R(A, B, C, D, E) A -> BC, CD -> E, B -> D, E -> A All possible candidate keys in above relation are {A, E, CD, BC} All attributes are on right sides of all functional dependencies are prime.
    - **Example 1 –** Find the highest normal form of a relation R(A,B,C,D,E) with FD set as {BC->D, AC->BE, B->E} Step 1. As we can see, (AC)+ ={A,C,B,E,D} but none of its subset can determine all attribute of relation, So AC will be candidate key. A or C can't be derived from any other attribute of the relation, so there will be only 1 candidate key {AC}. Step 2. Prime attributes are those attributes that are part of candidate key {A, C} in this example and others will be non-prime {B, D, E} in this example. Step 3. The relation R is in 1st normal form as a relational DBMS does not allow multi-valued or composite attribute. The relation is in 2nd normal form because BC->D is in 2nd normal form (BC is not a proper subset of candidate key AC) and AC->BE is in 2nd normal form (AC is candidate key) and B->E is in 2nd normal form (B is not a proper subset of candidate key AC). The relation is not in 3rd normal form because in BC->D (neither BC is a super key nor D is a prime attribute) and in B->E (neither B is a super key nor E is a prime attribute) but to satisfy 3rd normal for, either LHS of an FD should be super key or RHS should be prime attribute. So the highest normal form of relation will be 2nd Normal form.
    - **Example 2 –**For example consider relation R(A, B, C) A -> BC, B -> A and B both are super keys so above relation is in BCNF.

Applications of normal forms in DBMS:

Data consistency: Normal forms ensure that data is consistent and does not contain any redundant information. This helps to prevent inconsistencies and errors in the database.
Data redundancy: Normal forms minimize data redundancy by organizing data into tables that contain only unique data. This reduces the amount of storage space required for the database and makes it easier to manage.
Query performance: Normal forms can improve query performance by reducing the number of joins required to retrieve data. This helps to speed up query processing and improve overall system performance.
Database maintenance: Normal forms make it easier to maintain the database by reducing the amount of redundant data that needs to be updated, deleted, or modified. This helps to improve database management and reduce the risk of errors or inconsistencies.
Database design: Normal forms provide guidelines for designing databases that are efficient, flexible, and scalable. This helps to ensure that the database can be easily modified, updated, or expanded as needed.

1. BCNF is free from redundancy.
2. If a relation is in BCNF, then 3NF is also satisfied.
3. If all attributes of relation are prime attribute, then the relation is always in 3NF.
4. A relation in a Relational Database is always and at least in 1NF form.
5. Every Binary Relation (a Relation with only 2 attributes) is always in BCNF.
6. If a Relation has only singleton candidate keys(i.e. every candidate key consists of only 1 attribute), then the Relation is always in 2NF(because no Partial functional dependency possible).
7. Sometimes going for BCNF form may not preserve functional dependency. In that case go for BCNF only if the lost FD(s) is not required, else normalize till 3NF only.
8. There are many more Normal forms that exist after BCNF, like 4NF and more. But in real world database systems it’s generally not required to go beyond BCNF.

Lossless Join and Dependency Preserving Decomposition

Decomposition of a relation is done when a relation in relational model is not in appropriate normal form. Relation R is decomposed into two or more relations if decomposition is lossless join as well as dependency preserving.

Lossless Join Decomposition

If we decompose a relation R into relations R1 and R2,

- Decomposition is lossy if $R1 \bowtie R2 \supset R$
- Decomposition is lossless if $R1 \bowtie R2 = R$

To check for lossless join decomposition using FD set, following conditions must hold:

1. Union of Attributes of R1 and R2 must be equal to attribute of R. Each attribute of R must be either in R1 or in R2.

```
Att(R1) U Att(R2) = Att(R)
```

1. Intersection of Attributes of R1 and R2 must not be NULL.

```
Att(R1) ∩ Att(R2) ≠ Φ
```

1. Common attribute must be a key for at least one relation (R1 or R2)

```
Att(R1) ∩ Att(R2) -> Att(R1) or Att(R1) ∩ Att(R2) -> Att(R2)
```

For Example, A relation R (A, B, C, D) with FD set{A->BC} is decomposed into R1(ABC) and R2(AD) which is a lossless join decomposition as:

1. First condition holds true as Att(R1) U Att(R2) = (ABC) U (AD) = (ABCD) = Att(R).
2. Second condition holds true as Att(R1) ∩ Att(R2) = (ABC) ∩ (AD) ≠ Φ
3. Third condition holds true as Att(R1) ∩ Att(R2) = A is a key of R1(ABC) because A->BC is given.

Dependency Preserving Decomposition

If we decompose a relation R into relations R1 and R2, All dependencies of R either must be a part of R1 or R2 or must be derivable from combination of FD's of R1 and R2. For Example, A relation R (A, B, C, D) with FD set{A->BC} is decomposed into R1(ABC) and R2(AD) which is dependency preserving because FD A->BC is a part of R1(ABC). **GATE Question: Consider a schema R(A,B,C,D) and functional dependencies A->B and C->D. Then the decomposition of R into R1(AB) and R2(CD) is [GATE-CS-2001]** A. dependency preserving and lossless join B. lossless join but not dependency preserving C. dependency preserving but not lossless join D. not dependency preserving and not lossless join **Answer:** For lossless join decomposition, these three conditions must hold true:

1. Att(R1) U Att(R2) = ABCD = Att(R)
2. Att(R1) ∩ Att(R2) = Φ, which violates the condition of lossless join decomposition. Hence the decomposition is not lossless.

For dependency preserving decomposition, A->B can be ensured in R1(AB) and C->D can be ensured in R2(CD). Hence it is dependency preserving decomposition. So, the correct option is C.

Advantages of Lossless Join and Dependency Preserving Decomposition:

Improved Data Integrity: Lossless join and dependency preserving decomposition help to maintain the data integrity of the original relation by ensuring that all dependencies are preserved.
Reduced Data Redundancy: These techniques help to reduce data redundancy by breaking down a relation into smaller, more manageable relations.
Improved Query Performance: By breaking down a relation into smaller, more focused relations, query performance can be improved.
Easier Maintenance and Updates: The smaller, more focused relations are easier to maintain and update than the original relation, making it easier to modify the database schema and update the data.

Better Flexibility: Lossless join and dependency preserving decomposition can improve the flexibility of the database system by allowing for easier modification of the schema.

Disadvantages of Lossless Join and Dependency Preserving Decomposition:

Increased Complexity: Lossless join and dependency preserving decomposition can increase the complexity of the database system, making it harder to understand and manage.
Costly: Decomposing relations can be costly, especially if the database is large and complex. This can require additional resources, such as hardware and personnel.
Reduced Performance: Although query performance can be improved in some cases, in others, lossless join and dependency preserving decomposition can result in reduced query performance due to the need for additional join operations.
Limited Scalability: These techniques may not scale well in larger databases, as the number of smaller, focused relations can become unwieldy.

Multivalued Dependency (MVD) in DBMS

MVD or multivalued dependency means that for a single value of attribute ‘a’ multiple values of attribute ‘b’ exist. We write it as,

```
a --> --> b
```

It is read as: a is multi-valued dependent on b.

Suppose a person named Geeks is working on 2 projects Microsoft and Oracle and has 2 hobbies namely Reading and Music. This can be expressed in a tabular format in the following way.

	a	b	c
	NAME	PROJECT	HOBBY
t1	Geeks	MS	Reading
t2	Geeks	Oracle	Music
t3	Geeks	MS	Music
t4	Geeks	Oracle	Reading

Here project and hobby are multivalued attributes because they contain different values for the same name(Geeks)

Attributes(columns): a,b,c
Tupples(rows):t1,t2,t3,t4

R=set of attributes r=relation

Project and Hobby are multivalued attributes as they have more than one value for a single person i.e., Geeks.

Multi Valued Dependency (MVD) :
We can say that multivalued dependency exists if the following conditions are met.
Conditions for MVD :
Any attribute say **a** multiple define another attribute b; if any legal relation r(R), for all pairs of tuples t1 and t2 in r, such that,
`t1[a] = t2[a]`

Then there exists t3 and t4 in r such that.

`t1[a] = t2[a] = t3[a] = t4[a]`

`t1[b] = t3[b]; t2[b] = t4[b]`

`t1 = t4; t2 = t3`

Then multivalued (MVD) dependency exists.

To check the MVD in given table, we apply the conditions stated above and we check it with the values in the given table.

	a	b	c
	NAME	PROJECT	HOBBY
t1	Geeks	MS	Reading
t2	Geeks	Oracle	Music
t3	Geeks	MS	Music
t4	Geeks	Oracle	Reading

Condition-1 for MVD –
`t1[a] = t2[a] = t3[a] = t4[a]`

Finding from table,

`t1[a] = t2[a] = t3[a] = t4[a] = Geeks`

So, condition 1 is Satisfied.

Condition-2 for MVD –
`t1[b] = t3[b]`

`And`

`t2[b] = t4[b]`

Finding from table,

`t1[b] = t3[b] = MS`

`And`

`t2[b] = t4[b] = Oracle`

So, condition 2 is Satisfied.

Condition-3 for MVD –
∃c ∈ R-(a ∪ b) where R is the set of attributes in the relational table.

t1 = t4

And

t2=t3

Finding from table,

t1 = t4 = Reading

And

t2 = t3 = Music

So, condition 3 is Satisfied.

All conditions are satisfied, therefore,

a --> --> b

According to table we have got,

name --> --> project

And for,

a --> --> C

We get,

name --> --> hobby

Hence, we know that MVD exists in the above table and it can be stated by,

name --> --> project

name --> --> hobby

Introduction of 4th and 5th Normal form in DBMS

If two or more independent relation are kept in a single relation or we can say **multivalue dependency** occurs when the presence of one or more rows in a table implies the presence of one or more other rows in that same table. Put another way, two attributes (or columns) in a table are independent of one another, but both depend on a third attribute. A **multivalued dependency** always requires at least three attributes because it consists of at least two attributes that are dependent on a third.

For a dependency A -> B, if for a single value of A, multiple value of B exists, then the table may have multi-valued dependency. The table should have at least 3 attributes and B and C should be independent for A ->> B multivalued dependency. For example,

Person	Mobile	Food_Likes
Mahesh	9893/9424	Burger / pizza
Ramesh	9191	Pizza

```
Person->-> mobile,

Person ->-> food_likes
```

This is read as “person multidetermines mobile” and “person multidetermines food_likes.”

Note that a functional dependency is a special case of multivalued dependency. In a functional dependency X -> Y, every x determines exactly one y, never more than one.

Fourth normal form (4NF):
Fourth normal form (4NF) is a level of database normalization where there are no non-trivial multivalued dependencies other than a candidate key. It builds on the first three normal forms (1NF, 2NF and 3NF) and the Boyce-Codd Normal Form (BCNF). It states that, in addition to a database meeting the requirements of BCNF, it must not contain more than one multivalued dependency.

Properties – A relation R is in 4NF if and only if the following conditions are satisfied:

1. It should be in the Boyce-Codd Normal Form (BCNF).
2. the table should not have any Multi-valued Dependency.

A table with a multivalued dependency violates the normalization standard of Fourth Normal Form (4NK) because it creates unnecessary redundancies and can contribute to inconsistent data. To bring this up to 4NF, it is necessary to break this information into two tables.

Example – Consider the database table of a class which has two relations R1 contains student ID(SID) and student name (SNAME) and R2 contains course id(CID) and course name (CNAME).
Table – R1(SID, SNAME)

SID	SNAME
S1	A
S2	B

Table – R2(CID, CNAME)

CID	CNAME
C1	C
C2	D

When there cross product is done it resulted in multivalued dependencies:

Table – R1 X R2

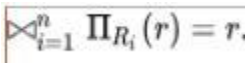
SID	SNAME	CID	CNAME
S1	A	C1	C
S1	A	C2	D
S2	B	C1	C
S2	B	C2	D

Multivalued dependencies (MVD) are:

```
SID->->CID; SID->->CNAME; SNAME->->CNAME
```

Joint dependency – Join decomposition is a further generalization of Multivalued dependencies. If the join of R1 and R2 over C is equal to relation R then we can say that a join
dependency (JD) exists, where R1 and R2 are the decomposition R1(A, B, C) and R2(C, D) of a given relations R (A, B, C, D). Alternatively, R1 and R2 are a lossless decomposition of R. A JD ⋈ {R1, R2, …, Rn} is said to hold over a relation R if R1, R2, ….., Rn is a lossless-join decomposition. The *(A, B, C, D), (C, D) will be a JD of R if the join of join's attribute is equal to

the relation R. Here, *(R1, R2, R3) is used to indicate that relation R1, R2, R3 and so on are a JD of R.
Let R is a relation schema R1, R2, R3……..Rn be the decomposition of R. r(R) is said to satisfy join dependency if and only if



Example –
Table – R1

Company	Product
C1	pendrive
C1	mic
C2	speaker
C2	speaker

```
Company->->Product
```

Table – R2

Agent	Company
Aman	C1
Aman	C2
Mohan	C1

```
Agent->->Company
```

Table – R3

Agent	Product
Aman	pendrive
Aman	mic

Agent	Product
Aman	speaker
Mohan	speaker

`Agent->->Product`

Table – R1⋈R2⋈R3

Company	Product	Agent
C1	pendrive	Aman
C1	mic	Aman
C2	speaker	speaker
C1	speaker	Aman

`Agent->->Product`

Fifth Normal Form / Projected Normal Form (5NF):
A relation R is in 5NF if and only if every join dependency in R is implied by the candidate keys of R. A relation decomposed into two relations must have loss-less join Property, which ensures that no spurious or extra tuples are generated, when relations are reunited through a natural join.

Properties – A relation R is in 5NF if and only if it satisfies following conditions:

1. R should be already in 4NF.

2. It cannot be further non loss decomposed (join dependency)

Example – Consider the above schema, with a case as "if a company makes a product and an agent is an agent for that company, then he always sells that product for the company". Under these circumstances, the ACP table is shown as:

Table – ACP

Agent	Company	Product
A1	PQR	Nut

Agent	Company	Product
A1	PQR	Bolt
A1	XYZ	Nut
A1	XYZ	Bolt
A2	PQR	Nut

The relation ACP is again decompose into 3 relations. Now, the natural Join of all the three relations will be shown as:

Table – R1

Agent	Company
A1	PQR
A1	XYZ
A2	PQR

Table – R2

Agent	Product
A1	Nut
A1	Bolt
A2	Nut

Table – R3

Company	Product
PQR	Nut

Company	Product
PQR	Bolt
XYZ	Nut
XYZ	Bolt

Result of Natural Join of R1 and R3 over ‘Company’ and then Natural Join of R13 and R2 over ‘Agent’and ‘Product’ will be table **ACP**.
Hence, in this example, all the redundancies are eliminated, and the decomposition of ACP is a lossless join decomposition. Therefore, the relation is in 5NF as it does not violate the property of lossless join.

Domain Key Normal Form in DBMS

Prerequisites – Normal Forms, 4th and 5th Normal form, find the highest normal form of a relation It is basically a process in database to organize data efficiently. Basically there are two goals of doing normalization these are as follows:

1. To remove repeated data or in simple words we can say to remove redundant data.
2. Second one is to ensure that there will be data dependencies.

Steps can be done to achieve Normalization:

1. Remove repeating groups or to eliminate repeating groups.
2. Eliminate or remove repeating data.
3. Remove those columns that are not dependent on Key.
4. Multiple relationship should be isolated independently.
5. Isolate Semantically Related Multiple Relationships

There are several types of normal forms, a lower numbered normal form always weaker than the higher numbered normal form. For example, 1st normal form is weaker than that of 2nd normal form. These are as: 1st, 2nd, 3rd, Boyce-code normal form, 4th, 5th, and domain key normal form. But, in this article, we will discuss only about Domain-Key Normal Form. **Domain key normal form (DKNF) –** There is no Hard and fast rule to define normal form up to 5NF. Historically the process of normalization and the process of discovering undesirable dependencies were carried through 5NF, but it has been possible to define the stricter normal form that takes into account additional type of dependencies and constraints. The basic idea behind the *DKNF* is to specify the normal form that takes into account all the possible dependencies and constraints. In simple words, we can say that DKNF is a normal form used in database normalization which requires that the database contains no constraints other than domain constraints and key constraints. In other words, a relation schema is said to be in DKNF only if all the constraints and dependencies that should hold on the valid relation state can be enforced

simply by enforcing the domain constraints and the key constraints on the relation. For a relation in DKNF, it becomes very straight forward to enforce all the database constraints by simply checking that each attribute value is a tuple is of the appropriate domain and that every key constraint is enforced. Reason to use DKNF are as follows:

1. To avoid general constraints in the database that are not clear key constraints.
2. Most database can easily test or check key constraints on attributes.

However, because of the difficulty of including complex constraints in a DKNF relation its practical utility is limited means that they are not in practical use, since it may be quite difficult to specify general integrity constraints. Let's understand this by taking an example: **Example –** Consider relations CAR (MAKE, vin#) and MANUFACTURE (vin#, country), Where vin# represents the vehicle identification number 'country' represents the name of the country where it is manufactured. A general constraint may be of the following form: If the MAKE is either 'HONDA' or 'MARUTI' then the first character of the vin# is a 'B' If the country of manufacture is 'INDIA' If the MAKE is 'FORD' or 'ACCURA', the second character of the vin# is a 'B" if the country of manufacture is 'INDIA'. There is no simplified way to represent such constraints short of writing a procedure or general assertion to test them. Hence such a procedure needs to enforce an appropriate integrity constraint. However, transforming a higher normal form into domain/key normal form is not always a dependency-preserving transformation and these are not possible always.

Advantages of Domain Key Normal Form:

Improved Data Integrity: DK/NF ensures that all dependencies and constraints are preserved, resulting in improved data integrity.
Reduced Data Redundancy: DK/NF reduces data redundancy by breaking down a relation into smaller, more focused relations.
Improved Query Performance: By breaking down a relation into smaller, more focused relations, query performance can be improved.
Easier Maintenance and Updates: The smaller, more focused relations are easier to maintain and update than the original relation, making it easier to modify the database schema and update the data.
Better Flexibility: DK/NF can improve the flexibility of the database system by allowing for easier modification of the schema.

Disadvantages of Domain Key Normal Form:

Increased Complexity: Normalizing a relation to DK/NF can increase the complexity of the database system, making it harder to understand and manage.
Costly: Normalizing a relation to DK/NF can be costly, especially if the database is large and complex. This can require additional resources, such as hardware and personnel.

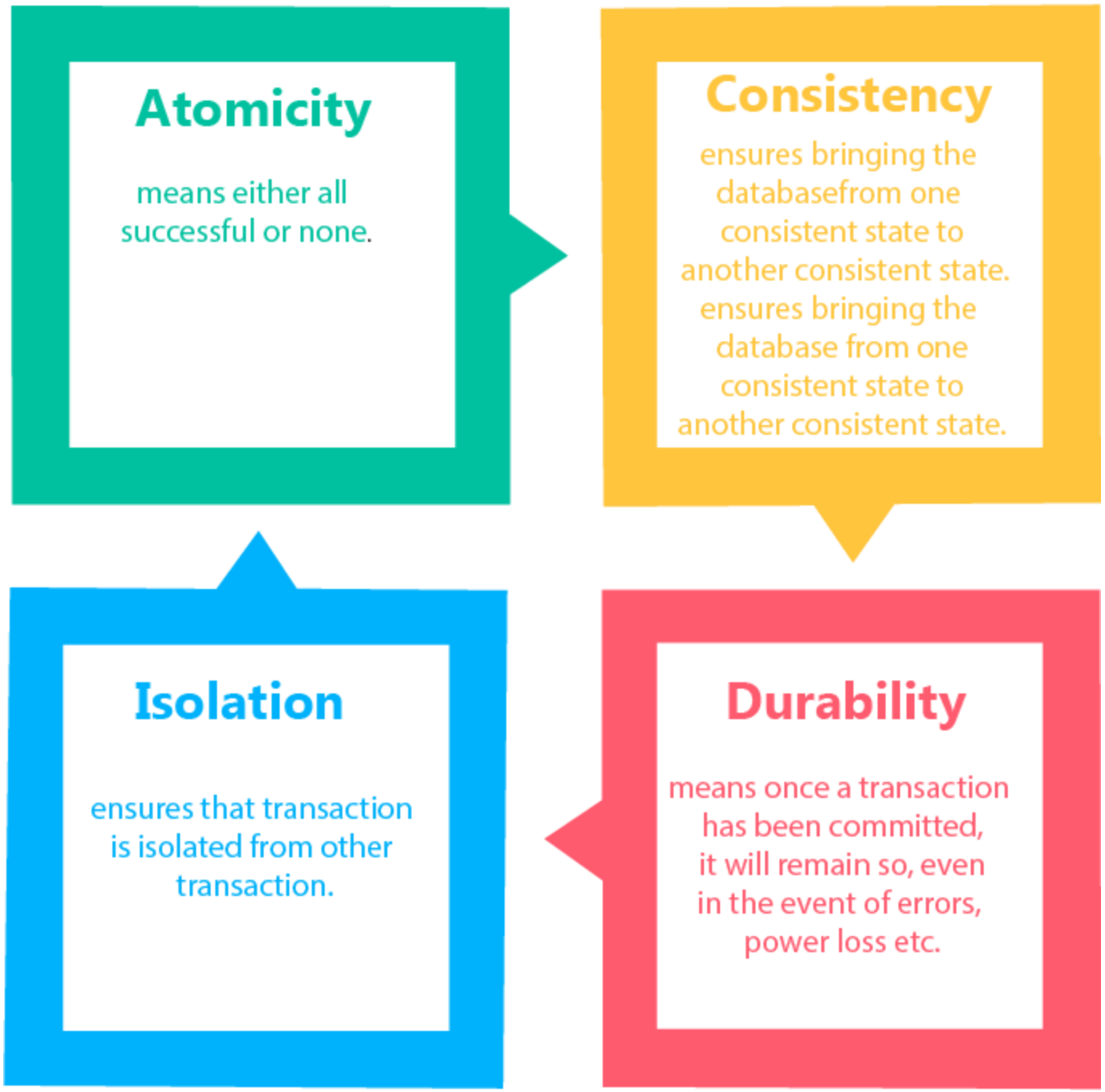
Reduced Performance: Although query performance can be improved in some cases, in others, normalization to DK/NF can result in reduced query performance due to the need for additional join operations.
Limited Scalability: Normalization to DK/NF may not scale well in larger databases, as the number of smaller, focused relations can become unwieldy.

Transaction property

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

Property of Transaction

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



Atomicity

- It states that all operations of the transaction take place at once if not, the transaction is aborted.

- There is no midway, i.e., the transaction cannot occur partially. Each transaction is treated as one unit and either run to completion or is not executed at all.

Atomicity involves the following two operations:

Abort: If a transaction aborts then all the changes made are not visible.

Commit: If a transaction commits then all the changes made are visible.

Example: Let's assume that following transaction T consisting of T1 and T2. A consists of Rs 600 and B consists of Rs 300. Transfer Rs 100 from account A to account B.

T1	T2
Read(A) A:= A-100 Write(A)	Read(B) Y:= Y+100 Write(B)

After completion of the transaction, A consists of Rs 500 and B consists of Rs 400.

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

Consistency

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

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

1. Total before T occurs = 600+300=900
2. Total after T occurs= 500+400=900

Therefore, the database is consistent. In the case when T1 is completed but T2 fails, then inconsistency will occur.

Isolation

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

Durability

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

Transaction Management

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

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

Steps for ATM Transaction

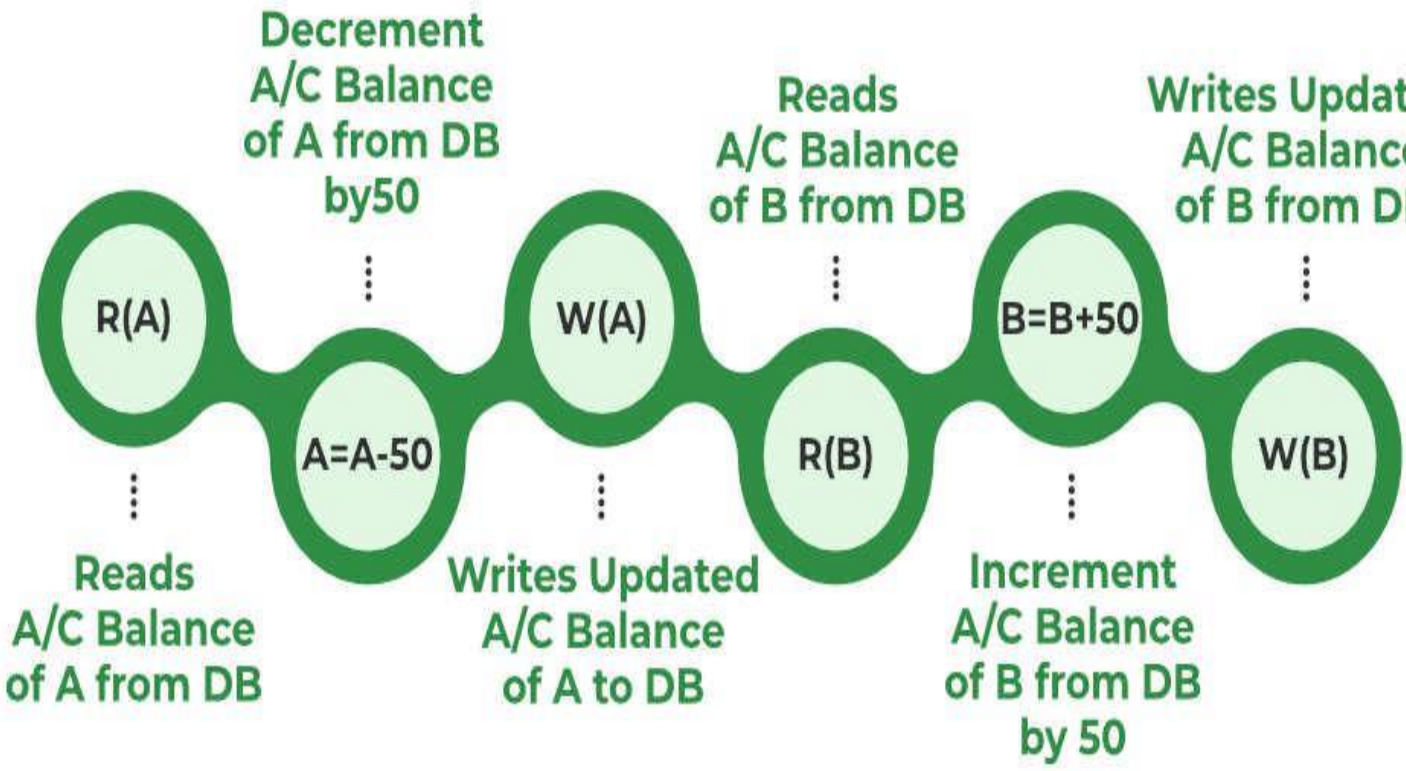
1. Transaction Start.
2. Insert your ATM card.
3. Select a language for your transaction.
4. Select the Savings Account option.
5. Enter the amount you want to withdraw.
6. Enter your secret pin.
7. Wait for some time for processing.
8. Collect your Cash.
9. Transaction Completed.

A transaction can include the following basic database access operation.

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

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

```
R(A) -- 500        // Accessed from RAM.

A = A-50           // Deducting 50₹ from A.

W(A)--450          // Updated in RAM.

R(B) -- 800        // Accessed from RAM.

B=B+50             // 50₹ is added to B's Account.

W(B) --850         // Updated in RAM.

commit             // The data in RAM is taken back to Hard Disk.
```



Stages of Transaction

Note: The updated value of Account A = 450₹ and Account B = 850₹.
All instructions before committing come under a partially committed state and are stored in RAM. When the commit is read the data is fully accepted and is stored on Hard Disk.

If the transaction is failed anywhere before committing we have to go back and start from the beginning. We can't continue from the same state. This is known as Roll Back.

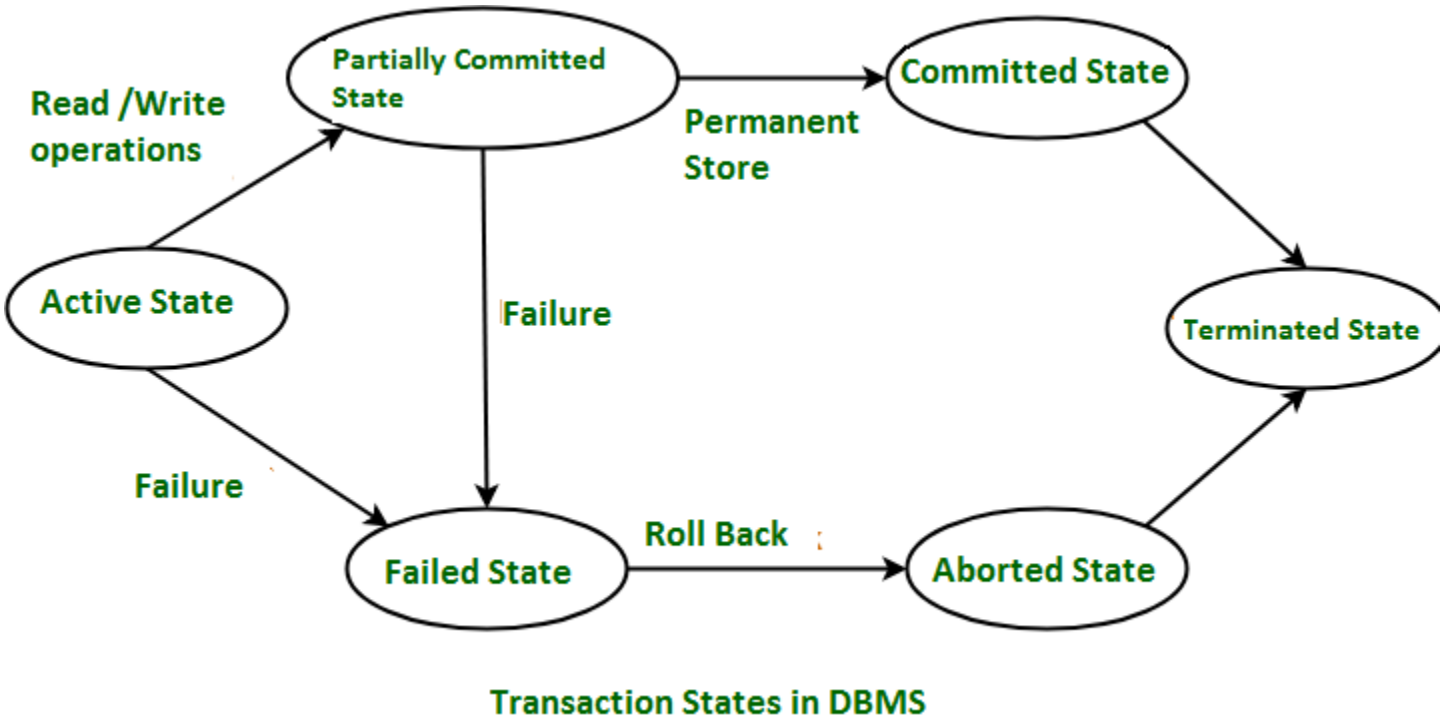
Desirable Properties of Transaction (ACID Properties)

For a transaction to be performed in DBMS, it must possess several properties often called ACID properties.

- **A** – Atomicity
- **C** – Consistency
- **I** – Isolation
- **D** – Durability

Transaction States

Transactions can be implemented using SQL queries and Servers. In the below-given diagram, you can see how transaction states work.



Transaction States

The transaction has the four properties. These are used to maintain consistency in a database, before and after the transaction.
Property of Transaction:
1. Atomicity
2. Consistency
3. Isolation
4. Durability

Atomicity:

- t states that all operations of the transaction take place at once if not, the transactions aborted.
- There is no midway, i.e., the transaction cannot occur partially. Each transaction is treated as one unit and either run to completion or is not executed at all.
- Atomicity involves the following two operations:
- ***Abort:*** If a transaction aborts, then all the changes made are not visible.
- ***Commit:*** If a transaction commits then all the changes made are visible.

Consistency:

- The integrity constraints are maintained so that the database is consistent before and after the transaction.

- The execution of a transaction will leave a database in either its prior stable state or anew stable state.
- The consistent property of database states that every transaction sees a consistent database instance.
- The transaction is used to transform the database from one consistent state to another consistent state.

Isolation:

- It shows that the data which is used at the time of execution of a transaction cannot be used by the second transaction until the first one is completed.
- In isolation, if the transaction T1 is being executed and using the data item X, then that data item can't be accessed by any other transaction T2 until the transaction T1ends.
- The concurrency control subsystem of the DBMS enforced the isolation property

Durability:

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

IMPLEMENTATION OF ATOMICITY AND DURABILITY:

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

E.g. the shadow-database scheme:

Shadow copy:

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

TRANSACTION ISOLATION LEVELS IN DBMS:

Some other transaction may also have used value produced by the failed transaction. So we also have to rollback those transactions.

The SQL standard defines four isolation levels :

1. Read Uncommitted – Read Uncommitted is the lowest isolation level. In this level,one transaction may read not yet committed changes made by other transaction, therebyallowing dirty reads. In this level, transactions are not isolated from each other.

2. **Read Committed** – This isolation level guarantees that any data read is committed atthe moment it is read. Thus it does not allows dirty read. The transaction holds a read orwrite lock on the current row, and thus prevent other transactions from reading,updating or deleting it.

3. **Repeatable Read** – This is the most restrictive isolation level. The transaction holdsead locks on all rows it references and writes locks on all rows it inserts, updates.deletes. Since other transaction cannot read, update or delete these rows, consequently it
avoids non-repeatable read.

4. Serializable – This is the Highest isolation level. A serializable execution is guaranteed to be serializable. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

FAILURE CLASSIFICATION:

To find that where the problem has occurred, we generalize a failure into the following categories:

1. Transaction failure
2. System crash
3. Disk failure

1. Transaction failure:

- The transaction failure occurs when it fails to execute or when it reaches a point from where it can't go any further. If a few transactions 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.

2. Syntax error: 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.

2. System Crash:

- System failure can occur due to power failure or other hardware or software failure. Example: Operating system error.

Fail-stop assumption: In the system crash, non-volatile storage is assumed not to be corrupted.

3. Disk Failure:

- It occurs where hard-disk drives or storage drives used to fail frequently. It was a common problem in the early days of technology evolution.
- Disk failure occurs due to the formation of bad sectors, disk head crash, and unreachability to the disk or any other failure, which destroy all or part of disk storage.

Serializability:

It is an important aspect of Transactions. In simple meaning, you can say that serializability is a way to check whether two transactions working on a database are maintaining database consistency or not.

It is of two types:

1. Conflict Serializability
2. View Serializability

Schedule

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.

It is of two types:

1. Serial Schedule
2. Non-Serial Schedule

Uses of Transaction Management

- The DBMS is used to schedule the access of data concurrently. It means that the user can access multiple data from the database without being interfered with by each other. Transactions are used to manage concurrency.
- It is also used to satisfy ACID properties.
- It is used to solve Read/Write Conflicts.
- It is used to implement Recoverability, Serializability, and Cascading.
- Transaction Management is also used for Concurrency Control Protocols and the Locking of data.

Disadvantages of using a Transaction

- It may be difficult to change the information within the transaction database by end-users.
- We need to always roll back and start from the beginning rather than continue from the previous state.

Serializability in DBMS

The backbone of the most modern application is the form of DBMS. When we design the form properly, then it provides high-performance and relative storage solutions to our application. In this topic, we are going to explain the serializability concept and how this concept affects the DBMS deeply. We also understand the concept of serializability with some examples. Finally, we will conclude this topic with an example of the importance of serializability.

What is Serializability in DBMS?

In the field of computer science, serializability is a term that is a property of the system that describes how the different process operates the shared data. If the result given by

the system is similar to the operation performed by the system, then in this situation, we call that system serializable. Here the cooperation of the system means there is no overlapping in the execution of the data. In DBMS, when the data is being written or read then, the DBMS can stop all the other processes from accessing the data.

In the MongoDB developer certificate, the DBMS uses various locking systems to allow the other processes while maintaining the integrity of the data. In MongoDB, the most restricted level for serializability is the employee can be restricted by two-phase locking or 2PL. In the first phase of the locking level, the data objects are locked before the execution of the operation. When the transaction has been accomplished, then the lock for the data object is released. This process guarantees that there is no conflict in operation and that all the transaction views the database as a conflict database.The two-phase locking or 2PL system provides a strong guarantee for the conflict of the database.

It can reduce the decreased performance and then increase the overhead acquiring capacity and then release the lock of the data. As a result, the system allows the constraint serializability for better performance of the DBMS. This ensures that the final result is the same as some sequential execution and performs the improvement of the operation that is involved in the database.

Thus, serializability is the system's property that describes how the different process operates the shared data. In DBMS, the overall Serializable property is adopted by locking the data during the execution of other processes. Also, serializability ensures that the final result is equivalent to the sequential operation of the data.

What is a Serializable Schedule?

- In DBMS, the Serializable schedule is a property in which the read and write operation sequence does not disturb the serializability property. This property ensures that the transaction is executed automatically with the other transaction. In DBMS, the order of the serializability must be the same as some serial schedules of the same transaction.
- In DBMS, there are several algorithms available to check the serializability of the database. One of the most important algorithms is the conflict serializability algorithm. The conflict serializability algorithm is the ability to check the potential of the conflict in the database. When the two transactions access the same data, this conflict occurs in the database. If there is no conflict, then there is

guaranteed serializability in the database. However, if there is a conflict occurs, then there is a chance of serializability.

- Another algorithm that is used to check the serializability of the database is the DBMS algorithm. With the help of the DBMS algorithm, we can check the potential mutual dependencies between two transactions. When the two transactions give the correct output, then mutual dependencies exist. When there are no mutual dependencies, then there is a guaranteed serializable in the database. However, if there are mutual dependencies, then there will be a chance of serializable.
- We can also check the serializability of the database by using the precedence graph algorithm. A precedence relationship exists when one transaction must precede another transaction for the schedule to be valid. If there are no cycles, then the serializability of schedules in DBMS is guaranteed. However, if there are cycles, the schedule may or may not be serializable. To understand different algorithms comprehensively, take the MongoDB Administration certification and get expert analysis on the concept of serializability in DBMS.

Types of Serializability

In DBMS, all the transaction should be arranged in a particular order, even if all the transaction is concurrent. If all the transaction is not serializable, then it produces the incorrect result.

In DBMS, there are different types of serializable. Each type of serializable has some advantages and disadvantages. The two most common types of serializable are view serializability and conflict serializability.

1. Conflict Serializability

Conflict serializability is a type of conflict operation in serializability that operates the same data item that should be executed in a particular order and maintains the consistency of the database. In DBMS, each transaction has some unique value, and every transaction of the database is based on that unique value of the database.

This unique value ensures that no two operations having the same conflict value are executed concurrently. For example, let's consider two examples, i.e., the order table and the customer table. One customer can have multiple orders, but each order only

belongs to one customer. There is some condition for the conflict serializability of the database. These are as below.

- Both operations should have different transactions.
- Both transactions should have the same data item.
- There should be at least one write operation between the two operations.

If there are two transactions that are executed concurrently, one operation has to add the transaction of the first customer, and another operation has added by the second operation. This process ensures that there would be no inconsistency in the database.

2. View Serializability

View serializability is a type of operation in the serializable in which each transaction should produce some result and these results are the output of proper sequential execution of the data item. Unlike conflict serialized, the view serializability focuses on preventing inconsistency in the database. In DBMS, the view serializability provides the user to view the database in a conflicting way.

In DBMS, we should understand schedules S1 and S2 to understand view serializability better. These two schedules should be created with the help of two transactions T1 and T2. To maintain the equivalent of the transaction each schedule has to obey the three transactions. These three conditions are as follows.

- The first condition is each schedule has the same type of transaction. The meaning of this condition is that both schedules S1 and S2 must not have the same type of set of transactions. If one schedule has committed the transaction but does not match the transaction of another schedule, then the schedule is not equivalent to each other.
- The second condition is that both schedules should not have the same type of read or write operation. On the other hand, if schedule S1 has two write operations while schedule S2 has one write operation, we say that both schedules are not equivalent to each other. We may also say that there is no problem if the number of the read operation is different, but there must be the same number of the write operation in both schedules.
- The final and last condition is that both schedules should not have the same conflict. Order of execution of the same data item. For example, suppose the

transaction of schedule S1 is T1, and the transaction of schedule S2 is T2. The transaction T1 writes the data item A, and the transaction T2 also writes the data item A. in this case, the schedule is not equivalent to each other. But if the schedule has the same number of each write operation in the data item then we called the schedule equivalent to each other.

Testing of Serializability in DBMS with Examples

Serializability is a type of property of DBMS in which each transaction is executed independently and automatically, even though these transactions are executed concurrently. In other words, we can say that if there are several transactions executed concurrently, then the main work of the serializability function is to arrange these several transactions in a sequential manner.

For better understanding, let's explain these with an example. Suppose there are two users Sona and Archita. Each executes two transactions. Let's transactions T1 and T2 are executed by Sona, and T3 and T4 are executed by Archita. Suppose transaction T1 reads and writes the data item A, transaction T2 reads the data item B, transaction T3 reads and writes the data item C and transaction T4 reads the data item D. Lets the schedule the above transaction as below.

1. •
2. T1: Read A → Write A→ Read B → Write B`
3. •
4. `T2: Read B → Write B`
5. •
6. `T3: Read C → Write C→ Read D → Write D`
7. • `T4: Read D → Write D

Let's first discuss why these transactions are not serializable.

In order for a schedule to be considered serializable, it must first satisfy the conflict serializability property. In our example schedule above, notice that Transaction 1 (T1) and Transaction 2 (T2) read data item B before either writing it. This causes a conflict between T1 and T2 because they are both trying to read and write the same data item concurrently. Therefore, the given schedule does not conflict with serializability.

However, there is another type of serializability called view serializability which our example does satisfy. View serializability requires that if two transactions cannot see each other's updates (i.e., one transaction cannot see the effects of another concurrent transaction), the schedule is considered to view serializable. In our example, Transaction 2 (T2) cannot see any updates made by Transaction 4 (T4) because they do not share common data items. Therefore, the schedule is viewed as serializable.

It's important to note that conflict serializability is a stronger property than view serializability because it requires that all potential conflicts be resolved before any updates are made (i.e., each transaction must either read or write each data item before any other transaction can write it). View serializability only requires that if two transactions cannot see each other's updates, then the schedule is view serializable & it doesn't matter whether or not there are potential conflicts between them.

All in all, both properties are necessary for ensuring correctness in concurrent transactions in a database management system.

Benefits of Serializability in DBMS

Below are the benefits of using the serializable in the database.

1. **Predictable execution:** In serializable, all the threads of the DBMS are executed at one time. There are no such surprises in the DBMS. In DBMS, all the variables are updated as expected, and there is no data loss or corruption.
2. **Easier to Reason about & Debug:** In DBMS all the threads are executed alone, so it is very easier to know about each thread of the database. This can make the debugging process very easy. So we don't have to worry about the concurrent process.
3. **Reduced Costs:** With the help of serializable property, we can reduce the cost of the hardware that is being used for the smooth operation of the database. It can also reduce the development cost of the software.
4. **Increased Performance:**In some cases, serializable executions can perform better than their non-serializable counterparts since they allow the developer to optimize their code for performance.

Concurrency Control Techniques

Concurrency control is provided in a database to:

- (i) enforce isolation among transactions.
- (ii) preserve database consistency through consistency preserving execution of transactions.
- (iii) resolve read-write and write-read conflicts.

Various concurrency control techniques are:

1. Two-phase locking Protocol
2. Time stamp ordering Protocol
3. Multi version concurrency control
4. Validation concurrency control

These are briefly explained below. **1. Two-Phase Locking Protocol:** Locking is an operation which secures: permission to read, OR permission to write a data item. Two phase locking is a process used to gain ownership of shared resources without creating the possibility of deadlock. The 3 activities taking place in the two phase update algorithm are:

(i). Lock Acquisition
(ii). Modification of Data
(iii). Release Lock

Two phase locking prevents deadlock from occurring in distributed systems by releasing all the resources it has acquired, if it is not possible to acquire all the resources required without waiting for another process to finish using a lock. This means that no process is ever in a state where it is holding some shared resources, and waiting for another process to release a shared resource which it requires. This means that deadlock cannot occur due to resource contention. A transaction in the Two Phase Locking Protocol can assume one of the 2 phases:

- **(i) Growing Phase:** In this phase a transaction can only acquire locks but cannot release any lock. The point when a transaction acquires all the locks it needs is called the Lock Point.
- **(ii) Shrinking Phase:** In this phase a transaction can only release locks but cannot acquire any.

2. Time Stamp Ordering Protocol: A timestamp is a tag that can be attached to any transaction or any data item, which denotes a specific time on which the transaction or the data item had been used in any way. A timestamp can be implemented in 2 ways. One is to directly assign the current value of the clock to the transaction or data item. The other is to attach the value of a logical counter that keeps increment as new timestamps are required. The timestamp of a data item can be of 2 types:

- **(i) W-timestamp(X):** This means the latest time when the data item X has been written into.
- **(ii) R-timestamp(X):** This means the latest time when the data item X has been read from. These 2 timestamps are updated each time a successful read/write operation is performed on the data item X.

3. Multiversion Concurrency Control: Multiversion schemes keep old versions of data item to increase concurrency. **Multiversion 2 phase locking:** Each successful write results in the creation of a new version of the data item written. Timestamps are used to label the versions. When a read(X) operation is issued, select an appropriate version of X based on the timestamp of the transaction. **4. Validation Concurrency Control:** The optimistic approach is based on the assumption that the majority of the database operations do not conflict. The optimistic approach requires neither locking nor time stamping techniques. Instead, a transaction is executed without restrictions until it is committed. Using an optimistic approach, each transaction moves through 2 or 3 phases, referred to as read, validation and write.

- **(i)** During read phase, the transaction reads the database, executes the needed computations and makes the updates to a private copy of the database values. All update operations of the transactions are recorded in a temporary update file, which is not accessed by the remaining transactions.
- **(ii)** During the validation phase, the transaction is validated to ensure that the changes made will not affect the integrity and consistency of the database. If the validation test is positive, the transaction goes to a write phase. If the validation test is negative, he transaction is restarted and the changes are discarded.
- **(iii)** During the write phase, the changes are permanently applied to the database.

Concurrency Control Techniques

Concurrency control is provided in a database to:

- (i) enforce isolation among transactions.
- (ii) preserve database consistency through consistency preserving execution of transactions.
- (iii) resolve read-write and write-read conflicts.

Various concurrency control techniques are:

```
1. Two-phase locking Protocol
2. Time stamp ordering Protocol
3. Multi version concurrency control
4. Validation concurrency control
```

These are briefly explained below. **1. Two-Phase Locking Protocol:** Locking is an operation which secures: permission to read, OR permission to write a data item. Two phase locking is a process used to gain ownership of shared resources without creating the possibility of deadlock. The 3 activities taking place in the two phase update algorithm are:

```
(i). Lock Acquisition
(ii). Modification of Data
(iii). Release Lock
```

Two phase locking prevents deadlock from occurring in distributed systems by releasing all the resources it has acquired, if it is not possible to acquire all the resources required without waiting for another process to finish using a lock. This means that no process is ever in a state where it is holding some shared resources, and waiting for another process

to release a shared resource which it requires. This means that deadlock cannot occur due to resource contention. A transaction in the Two Phase Locking Protocol can assume one of the 2 phases:

- **(i) Growing Phase:** In this phase a transaction can only acquire locks but cannot release any lock. The point when a transaction acquires all the locks it needs is called the Lock Point.
- **(ii) Shrinking Phase:** In this phase a transaction can only release locks but cannot acquire any.

2. Time Stamp Ordering Protocol: A timestamp is a tag that can be attached to any transaction or any data item, which denotes a specific time on which the transaction or the data item had been used in any way. A timestamp can be implemented in 2 ways. One is to directly assign the current value of the clock to the transaction or data item. The other is to attach the value of a logical counter that keeps increment as new timestamps are required. The timestamp of a data item can be of 2 types:

- **(i) W-timestamp(X):** This means the latest time when the data item X has been written into.
- **(ii) R-timestamp(X):** This means the latest time when the data item X has been read from. These 2 timestamps are updated each time a successful read/write operation is performed on the data item X.

3. Multiversion Concurrency Control: Multiversion schemes keep old versions of data item to increase concurrency. **Multiversion 2 phase locking:** Each successful write results in the creation of a new version of the data item written. Timestamps are used to label the versions. When a read(X) operation is issued, select an appropriate version of X based on the timestamp of the transaction. **4. Validation Concurrency Control:** The optimistic approach is based on the assumption that the majority of the database operations do not conflict. The optimistic approach requires neither locking nor time stamping techniques. Instead, a transaction is executed without restrictions until it is committed. Using an optimistic approach, each transaction moves through 2 or 3 phases, referred to as read, validation and write.

- **(i)** During read phase, the transaction reads the database, executes the needed computations and makes the updates to a private copy of the database values. All update operations of the transactions are recorded in a temporary update file, which is not accessed by the remaining transactions.
- **(ii)** During the validation phase, the transaction is validated to ensure that the changes made will not affect the integrity and consistency of the database. If the validation test is positive, the transaction goes to a write phase. If the validation test is negative, he transaction is restarted and the changes are discarded.
- **(iii)** During the write phase, the changes are permanently applied to the database.

Concurrency Control Techniques

Concurrency control is provided in a database to:

- (i) enforce isolation among transactions.
- (ii) preserve database consistency through consistency preserving execution of transactions.
- (iii) resolve read-write and write-read conflicts.

Various concurrency control techniques are:

1. Two-phase locking Protocol
2. Time stamp ordering Protocol
3. Multi version concurrency control
4. Validation concurrency control

These are briefly explained below. **1. Two-Phase Locking Protocol:** Locking is an operation which secures: permission to read, OR permission to write a data item. Two phase locking is a process used to gain ownership of shared resources without creating the possibility of deadlock. The 3 activities taking place in the two phase update algorithm are:

(i). Lock Acquisition
(ii). Modification of Data
(iii). Release Lock

Two phase locking prevents deadlock from occurring in distributed systems by releasing all the resources it has acquired, if it is not possible to acquire all the resources required without waiting for another process to finish using a lock. This means that no process is ever in a state where it is holding some shared resources, and waiting for another process to release a shared resource which it requires. This means that deadlock cannot occur due to resource contention. A transaction in the Two Phase Locking Protocol can assume one of the 2 phases:

- **(i) Growing Phase:** In this phase a transaction can only acquire locks but cannot release any lock. The point when a transaction acquires all the locks it needs is called the Lock Point.
- **(ii) Shrinking Phase:** In this phase a transaction can only release locks but cannot acquire any.

2. Time Stamp Ordering Protocol: A timestamp is a tag that can be attached to any transaction or any data item, which denotes a specific time on which the transaction or the data item had been used in any way. A timestamp can be implemented in 2 ways. One is to directly assign the current value of the clock to the transaction or data item. The other is to attach the value of a logical counter that keeps increment as new timestamps are required. The timestamp of a data item can be of 2 types:

- **(i) W-timestamp(X):** This means the latest time when the data item X has been written into.
- **(ii) R-timestamp(X):** This means the latest time when the data item X has been read from. These 2 timestamps are updated each time a successful read/write operation is performed on the data item X.

3. Multiversion Concurrency Control: Multiversion schemes keep old versions of data item to increase concurrency. **Multiversion 2 phase locking:** Each successful write

results in the creation of a new version of the data item written. Timestamps are used to label the versions. When a read(X) operation is issued, select an appropriate version of X based on the timestamp of the transaction. **4. Validation Concurrency Control:** The optimistic approach is based on the assumption that the majority of the database operations do not conflict. The optimistic approach requires neither locking nor time stamping techniques. Instead, a transaction is executed without restrictions until it is committed. Using an optimistic approach, each transaction moves through 2 or 3 phases, referred to as read, validation and write.

- **(i)** During read phase, the transaction reads the database, executes the needed computations and makes the updates to a private copy of the database values. All update operations of the transactions are recorded in a temporary update file, which is not accessed by the remaining transactions.
- **(ii)** During the validation phase, the transaction is validated to ensure that the changes made will not affect the integrity and consistency of the database. If the validation test is positive, the transaction goes to a write phase. If the validation test is negative, he transaction is restarted and the changes are discarded.
- **(iii)** During the write phase, the changes are permanently applied to the database.

Multiple Granularity

Let's start by understanding the meaning of granularity.

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

Multiple Granularity:

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

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

- The first level or higher level shows the entire database.

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

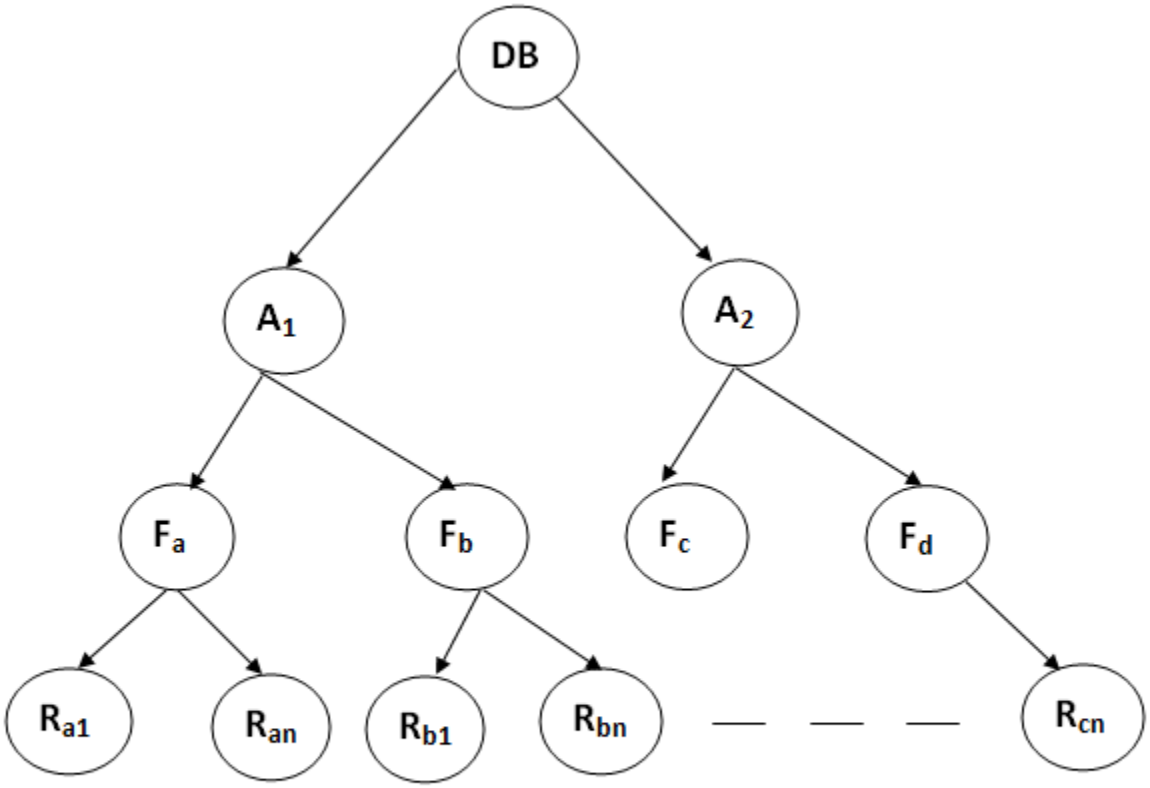


Figure: Multi Granularity tree Hierarchy

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

There are three additional lock modes with multiple granularity:

Intention Mode Lock

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

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

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

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

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

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

- Transaction T1 should follow the lock-compatibility matrix.
- Transaction T1 firstly locks the root of the tree. It can lock it in any mode.
- If T1 currently has the parent of the node locked in either IX or IS mode, then the transaction T1 will lock a node in S or IS mode only.
- If T1 currently has the parent of the node locked in either IX or SIX modes, then the transaction T1 will lock a node in X, SIX, or IX mode only.

- If T1 has not previously unlocked any node only, then the Transaction T1 can lock a node.
- If T1 currently has none of the children of the node-locked only, then Transaction T1 will unlock a node.

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

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

Deadlock Detection And Recovery

Deadlock detection and recovery is the process of detecting and resolving deadlocks in an operating system. A deadlock occurs when two or more processes are blocked, waiting for each other to release the resources they need. This can lead to a system-wide stall, where no process can make progress.

There are two main approaches to deadlock detection and recovery:

1. **Prevention**: The operating system takes steps to prevent deadlocks from occurring by ensuring that the system is always in a safe state, where deadlocks cannot occur. This is achieved through resource allocation algorithms such as the Banker's Algorithm.
2. **Detection and Recovery:** If deadlocks do occur, the operating system must detect and resolve them. Deadlock detection algorithms, such as the Wait-For Graph, are used to identify deadlocks, and recovery algorithms, such as the Rollback and Abort algorithm, are used to resolve them. The recovery algorithm releases the resources held by one or more processes, allowing the system to continue to make progress.

Difference Between Prevention and Detection/Recovery: Prevention aims to avoid deadlocks altogether by carefully managing resource allocation, while detection and recovery aim to identify and resolve deadlocks that have already occurred.

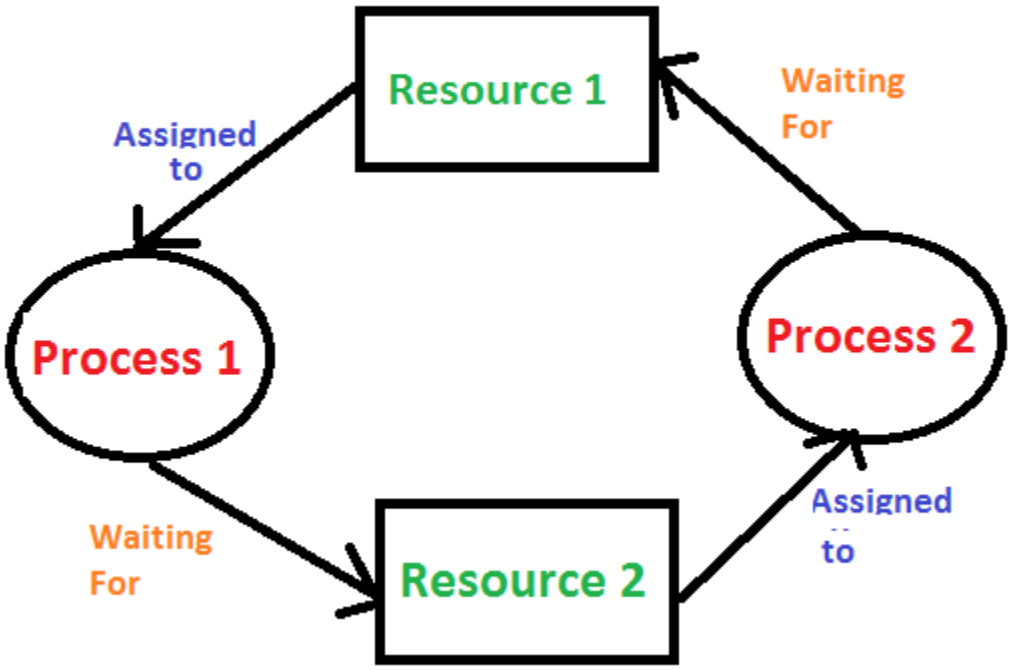
Deadlock detection and recovery is an important aspect of operating system design and management, as it affects the stability and performance of the system. The choice of deadlock detection and recovery approach depends on the specific requirements of the system and the trade-offs between performance, complexity, and risk tolerance. The operating system must balance these factors to ensure that deadlocks are effectively detected and resolved.

In the previous post, we discussed Deadlock Prevention and Avoidance. In this post, the Deadlock Detection and Recovery technique to handle deadlock is discussed.

Deadlock Detection :

1. If resources have a single instance –

In this case for Deadlock detection, we can run an algorithm to check for the cycle in the Resource Allocation Graph. The presence of a cycle in the graph is a sufficient condition for deadlock.



In the above diagram, resource 1 and resource 2 have single instances. There is a cycle R1 → P1 → R2 → P2. So, Deadlock is Confirmed.

2. If there are multiple instances of resources –

Detection of the cycle is necessary but not a sufficient condition for deadlock detection, in this case, the system may or may not be in deadlock varies according to different situations.

3. Wait-For Graph Algorithm –

The Wait-For Graph Algorithm is a deadlock detection algorithm used to detect deadlocks in a system where resources can have multiple instances. The algorithm works by constructing a Wait-For Graph, which is a directed graph that represents the dependencies between processes and resources.

Deadlock Recovery :
A traditional operating system such as Windows doesn't deal with deadlock recovery as it is a time and space-consuming process. Real-time operating systems use Deadlock recovery.

1. **Killing the process –**
   Killing all the processes involved in the deadlock. Killing process one by one. After killing each process check for deadlock again and keep repeating the process till the system recovers from deadlock. Killing all the processes one by one helps a system to break circular wait conditions.
2. **Resource Preemption –**
   Resources are preempted from the processes involved in the deadlock, and preempted resources are allocated to other processes so that there is a possibility of recovering the system from the deadlock. In this case, the system goes into starvation.
3. **Concurrency Control –** Concurrency control mechanisms are used to prevent data inconsistencies in systems with multiple concurrent processes. These mechanisms ensure that concurrent processes do not access the same data at the same time, which can lead to inconsistencies and errors. Deadlocks can occur in concurrent systems when two or more processes are blocked, waiting for each other to release the resources they need. This can result in a system-wide stall, where no process can make progress. Concurrency control mechanisms can help prevent deadlocks by managing access to shared resources and ensuring that concurrent processes do not interfere with each other.

ADVANTAGES OR DISADVANTAGES:

Advantages of Deadlock Detection and Recovery in Operating Systems:

1. **Improved System Stability:** Deadlocks can cause system-wide stalls, and detecting and resolving deadlocks can help to improve the stability of the system.
2. **Better Resource Utilization:** By detecting and resolving deadlocks, the operating system can ensure that resources are efficiently utilized and that the system remains responsive to user requests.
3. **Better System Design**: Deadlock detection and recovery algorithms can provide insight into the behavior of the system and the relationships between processes and resources, helping to inform and improve the design of the system.

Disadvantages of Deadlock Detection and Recovery in Operating Systems:

1. **Performance Overhead:** Deadlock detection and recovery algorithms can introduce a significant overhead in terms of performance, as the system must regularly check for deadlocks and take appropriate action to resolve them.
2. **Complexity:** Deadlock detection and recovery algorithms can be complex to implement, especially if they use advanced techniques such as the Resource Allocation Graph or Timestamping.
3. **False Positives and Negatives:** Deadlock detection algorithms are not perfect and may produce false positives or negatives, indicating the presence of deadlocks when they do not exist or failing to detect deadlocks that do exist.
4. **Risk of Data Loss**: In some cases, recovery algorithms may require rolling back the state of one or more processes, leading to data loss or corruption.

Database 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: The rollback/undo recovery technique is based on the principle of backing out or undoing the effects of a transaction that has not 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. 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 DBMS, and each technique has its own advantages and limitations that must be considered in the design of a recovery system

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 transaction are completed successfully and committed (the effect is recorded permanently in the database) or the transaction should have no effect on the database. There are both automatic and non-automatic ways for both, backing up of data and recovery from any failure situations. The techniques used to recover the lost data due to system crashes, transaction errors, viruses, catastrophic failure, incorrect commands execution, etc. are database recovery techniques. So to prevent data loss recovery techniques based on deferred update and immediate update 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: 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, 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 set 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.

- **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 another word, 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 another word, the database is restored using preserved data and valid transactions counted by their past saves.

Some of the backup techniques are as follows :

- **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 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.

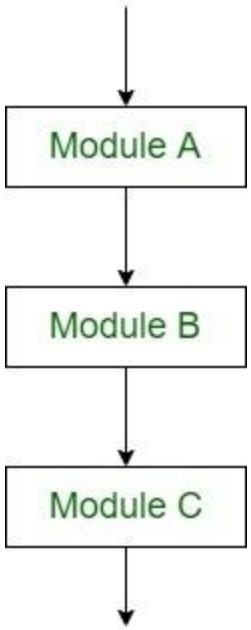
Control Structures in Programming Languages

Control Structures are just a way to specify flow of control in programs. Any algorithm or program can be more clear and understood if they use self-contained modules called as logic or control structures. It basically analyzes and chooses in which direction a program flows based on certain parameters or conditions. There are three basic types of logic, or flow of control, known as:

1. Sequence logic, or sequential flow
2. Selection logic, or conditional flow
3. Iteration logic, or repetitive flow

Let us see them in detail:

1. **Sequential Logic (Sequential Flow)**
   Sequential logic as the name suggests follows a serial or sequential flow in which the flow depends on the series of instructions given to the computer. Unless new instructions are given, the modules are executed in the obvious sequence. The sequences may be given, by means of numbered steps explicitly. Also, implicitly follows the order in which modules are written. Most of the processing, even some complex problems, will generally follow this elementary flow pattern.



Sequential Control flow

2. **Selection Logic (Conditional Flow)**
   Selection Logic simply involves a number of conditions or parameters which decides one out of several written modules. The structures which use these type of logic are known as **Conditional Structures**. These structures can be of three types:
   - **Single Alternative**This structure has the form:
   - `If (condition) then:`
   - `[Module A]`

     `[End of If structure]`

     Implementation:
     - C/C++ if statement with Examples
     - Java if statement with Examples
   - **Double Alternative**This structure has the form:
   - `If (Condition), then:`
   - `[Module A]`
   - `Else:`
   - `[Module B]`
   - `[End if structure]`

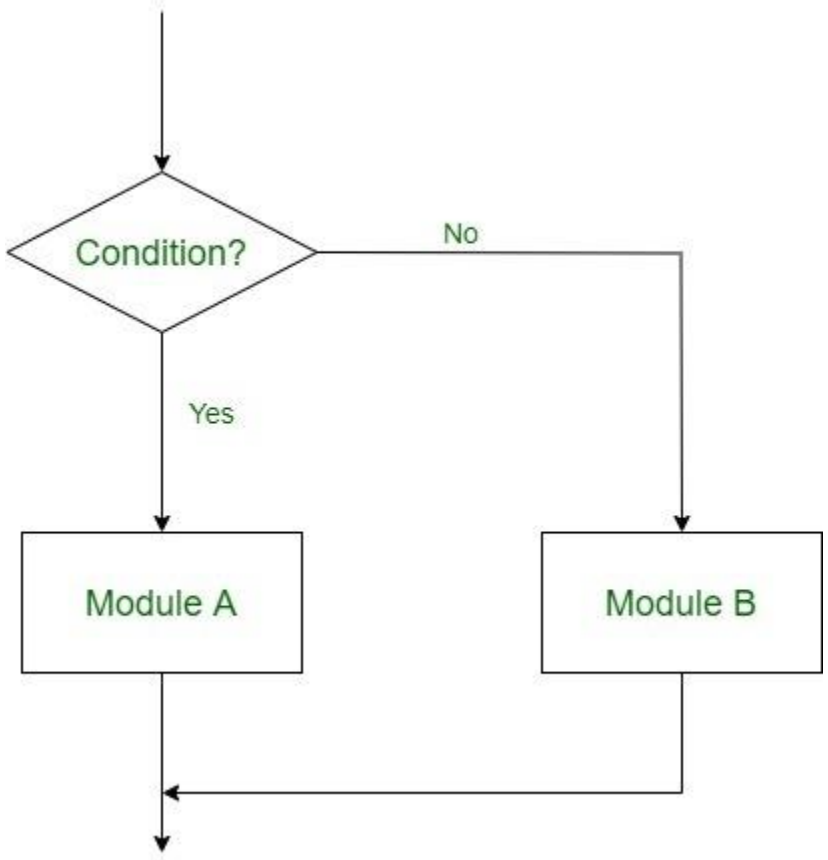
     Implementation:
     - C/C++ if-else statement with Examples
     - Java if-else statement with Examples
   - **Multiple Alternatives**This structure has the form:

- `If (condition A), then:`
- `      [Module A]`
- `Else if (condition B), then:`
- `      [Module B]`
- `         ..`
- `         ..`
- `Else if (condition N), then:`
- `      [Module N]`
- `[End If structure]`

Implementation:

- C/C++ if-else if statement with Examples
- Java if-else if statement with Examples

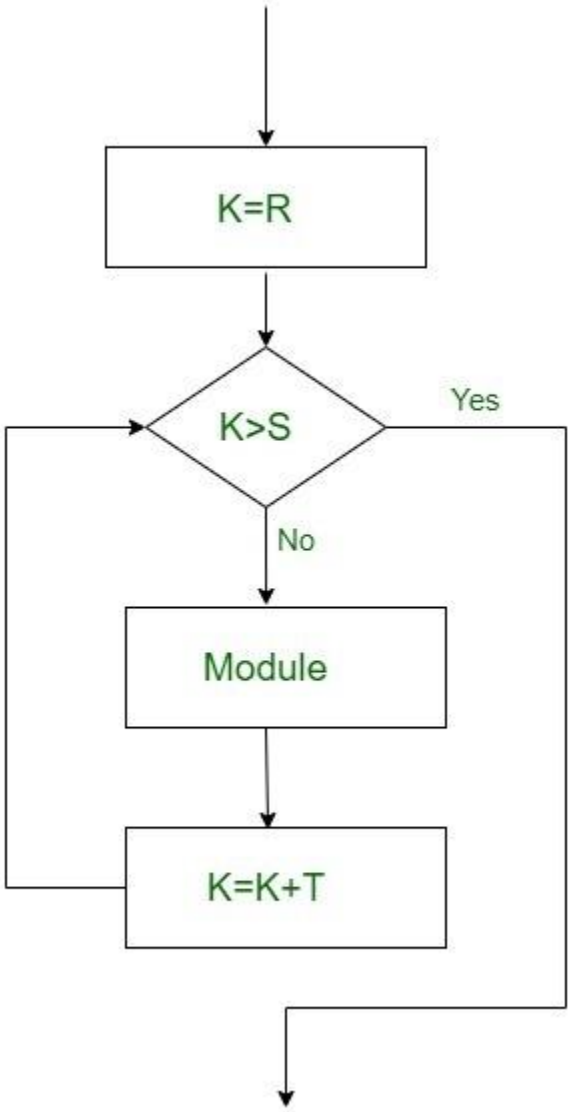
In this way, the flow of the program depends on the set of conditions that are written. This can be more understood by the following flow charts:



Double Alternative Control Flow

3. **Iteration Logic (Repetitive Flow)**
   The Iteration logic employs a loop which involves a repeat statement followed by a module known as the body of a loop.
   The two types of these structures are:
   - **Repeat-For Structure**
     This structure has the form:
   - `Repeat for i = A to N by I:`
   - `[Module]`
   - `[End of loop]`

     Here, A is the initial value, N is the end value and I is the increment. The loop ends when A>B. K increases or decreases according to the positive and negative value of I respectively.

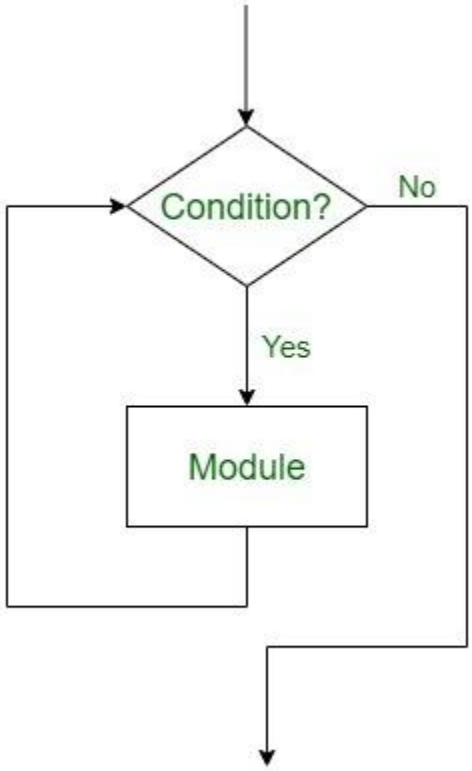


Repeat-For Flow

Implementation:

- C/C++ for loop with Examples
- Java for loop with Examples

- **Repeat-While Structure**
  It also uses a condition to control the loop. This structure has the form:
- `Repeat while condition:`
- `    [Module]`
- `[End of Loop]`



Repeat While Flow

Implementation:

- C/C++ while loop with Examples
- Java while loop with Examples

In this, there requires a statement that initializes the condition controlling the loop, and there must also be a statement inside the module that will change this condition leading to the end of the loop.

Exception Handling in PL/SQL

An exception is an error which disrupts the normal flow of program instructions. PL/SQL provides us the exception block which raises the exception thus helping the programmer to find out the fault and resolve it.

There are two types of exceptions defined in PL/SQL

1. User defined exception.
2. System defined exceptions.

Syntax to write an exception

```
WHEN exception THEN
    statement;
```

DECLARE
declarations section;

BEGIN
executable command(s);

EXCEPTION
WHEN exception1 THEN
statement1;
WHEN exception2 THEN
statement2;
[WHEN others THEN]
/ default exception handling code */*

END;

Note:
When other keyword should be used only at the end of the exception handling block as no exception handling part present later will get executed as the control will exit from the block after executing the WHEN OTHERS.

1. **System defined exceptions:**
   These exceptions are predefined in PL/SQL which get raised WHEN certain **database rule is violated.**
   System-defined exceptions are further divided into two categories:
   1. Named system exceptions.
   2. Unnamed system exceptions.
   - **Named system exceptions:** They have a predefined name by the system like ACCESS_INTO_NULL, DUP_VAL_ON_INDEX, LOGIN_DENIED etc. the list is quite big.
     So we will discuss some of the most commonly used exceptions:

     Lets create a table geeks.

     ```
     create table geeks(g_id int , g_name varchar(20), marks int);

     insert into geeks values(1, 'Suraj',100);

     insert into geeks values(2, 'Praveen',97);

     insert into geeks values(3, 'Jessie', 99);
     ```

G_id	G_name	marks
1	Suraj	100
2	Praveen	97
3	jessie	99

1. **NO_DATA_FOUND**: It is raised WHEN a SELECT INTO statement returns *no* rows. For eg:

```
DECLARE

   temp varchar(20);



BEGIN

   SELECT g_id into temp from geeks where
g_name='GeeksforGeeks';



exception

   WHEN no_data_found THEN

      dbms_output.put_line('ERROR');

      dbms_output.put_line('there is no name as');

      dbms_output.put_line('GeeksforGeeks in geeks
table');

end;
```

2. Output:
3. ERROR
4. there is no name as GeeksforGeeks in geeks table

5. **TOO_MANY_ROWS**:It is raised WHEN a SELECT INTO statement returns *more* than one row.

```
DECLARE

   temp varchar(20);



BEGIN



-- raises an exception as SELECT

-- into trying to return too many rows

   SELECT g_name  into temp from geeks;

   dbms_output.put_line(temp);



EXCEPTION

   WHEN too_many_rows  THEN

      dbms_output.put_line('error trying to SELECT too
many rows');



end;
```

6. Output:
7. `error trying to SELECT too many rows`
8. **VALUE_ERROR**:This error is raised WHEN a statement is executed that resulted in an arithmetic, numeric, string, conversion, or constraint error. This error mainly results from programmer error or invalid data input.

```
DECLARE

   temp number;



BEGIN

   SELECT g_name  into temp from geeks  where g_name='Suraj';

   dbms_output.put_line('the g_name is '||temp);



EXCEPTION

   WHEN value_error THEN

   dbms_output.put_line('Error');

   dbms_output.put_line('Change data type of temp to
varchar(20)');



END;
```

9. Output:
10. Error
11. Change data type of temp to varchar(20)
12. **ZERO_DIVIDE** = raises exception WHEN dividing with zero.

```
DECLARE

   a int:=10;
```

```
   b int:=0;

   answer int;



BEGIN

   answer:=a/b;

   dbms_output.put_line('the result after division
is'||answer);



exception

   WHEN zero_divide THEN

      dbms_output.put_line('dividing by zero please check
the values again');

      dbms_output.put_line('the value of a is '||a);

      dbms_output.put_line('the value of b is '||b);

END;
```

13. Output:

```
14.     dividing by zero please check the values again
15.     the value of a is 10
16.     the value of b is 0
```

2.

- **Unnamed system exceptions:**Oracle doesn't provide name for some system exceptions called unnamed system exceptions.These exceptions *don't* occur frequently.These exceptions have two parts *code and an associated message*. The way to handle to these exceptions is to *assign name* to them using **Pragma EXCEPTION_INIT**
  Syntax:

- `PRAGMA EXCEPTION_INIT(exception_name, -error_number);`

  error_number are pre-defined and have negative integer range from -20000 to -20999.

  Example:

```
DECLARE

   exp exception;

   pragma exception_init (exp, -20015);

   n int:=10;



BEGIN

   FOR i  IN 1..n LOOP

      dbms_output.put_line(i*i);

         IF i*i=36 THEN

            RAISE exp;

         END IF;

   END LOOP;



EXCEPTION

   WHEN exp  THEN

      dbms_output.put_line('Welcome to GeeksforGeeks');
```

```
END;
```

Output:

```
1
4
9
16
25
36
Welcome to GeeksforGeeks
```

3.

User defined exceptions:
This type of users can create their own exceptions according to the need and to raise these exceptions explicitly ***raise*** command is used.
Example:

- Divide non-negative integer x by y such that the result is greater than or equal to 1. From the given question we can conclude that there exist two exceptions
    - Division be zero.
    - If result is greater than or equal to 1 means y is less than or equal to x.

```
DECLARE

   x int:=&x; /*taking value at run time*/

   y int:=&y;

   div_r float;

   exp1 EXCEPTION;
```

```
   exp2 EXCEPTION;



BEGIN

   IF y=0 then

       raise exp1;



   ELSEIF y > x then

      raise exp2;



   ELSE

      div_r:= x / y;

      dbms_output.put_line('the result is '||div_r);



   END IF;



EXCEPTION

   WHEN exp1 THEN

      dbms_output.put_line('Error');

      dbms_output.put_line('division by zero not allowed');
```

```
   WHEN exp2 THEN

      dbms_output.put_line('Error');

      dbms_output.put_line('y is greater than x please check
the input');



END;
```

Input 1: x = 20
y = 10

Output: the result is 2
Input 2: x = 20
y = 0

Output:
Error
division by zero not allowed
Input 3: x=20
y = 30

Output:<.em>
Error
y is greater than x please check the input

RAISE_APPLICATION_ERROR*:*
It is used to display user-defined error messages with error number whose range is in between -20000 and -20999. When RAISE_APPLICATION_ERROR executes it returns error message and error code which looks ***same as Oracle built-in error.***
Example:

```
DECLARE

    myex EXCEPTION;

    n NUMBER :=10;
```

```
BEGIN

    FOR i  IN 1..n LOOP

    dbms_output.put_line(i*i);

        IF i*i=36 THEN

        RAISE myex;

        END IF;

    END LOOP;



EXCEPTION

    WHEN myex  THEN

        RAISE_APPLICATION_ERROR(-20015, 'Welcome to
GeeksForGeeks');



END;
```

Output:
Error report:
ORA-20015: Welcome to GeeksForGeeks
ORA-06512: at line 13

1
4
9
16
25
36

Note: *The output is based on Oracle Sql developer, the output order might change IF you're running this code somewhere else.*

Scope rules in exception handling*:*

1. *We can't DECLARE an exception twice but we can DECLARE the same exception in* ***two dIFferent blocks.***
2. *Exceptions DECLAREd inside a block are local to that block and global to all its sub-blocks.*

As a block can reference only local or global exceptions, enclosing blocks cannot reference exceptions DECLAREd in a sub-block.

If we reDECLARE a global exception in a sub-block, the local declaration prevails i.e. the scope of local is more.

Example:

```
DECLARE

   GeeksforGeeks EXCEPTION;

   age NUMBER:=16;

BEGIN



   --  sub-block BEGINs

   DECLARE



      -- this declaration prevails

      GeeksforGeeks  EXCEPTION;

      age NUMBER:=22;



   BEGIN
```

```
      IF age > 16 THEN

         RAISE GeeksforGeeks; /* this is not handled*/

      END IF;



   END;

   -- sub-block ends



EXCEPTION

  -- Does not handle raised exception

  WHEN GeeksforGeeks THEN

    DBMS_OUTPUT.PUT_LINE

      ('Handling  GeeksforGeeks exception.');



  WHEN OTHERS THEN

    DBMS_OUTPUT.PUT_LINE

      ('Could not recognize exception GeeksforGeeks  in this
scope.');

END;
```

Output:

Could not recognize exception GeeksforGeeks in this scope.

Advantages:

- *Exception handling is very useful for error handling, without it we have to issue the command at every point to check for execution errors:*

  Example:

- *Select ..*
- *.. check for 'no data found' error*
- *Select ..*
- *.. check for 'no data found' error*
- *Select ..*
- *.. check for 'no data found' error*

*Here we can see that it is not **robust** as error processing is not separated from normal processing and IF we miss some line in the code than it may lead to some other kind of error.*

- *With exception handling we handle errors without writing statements multiple times and we can even handle **dIFferent** types of errors in one exception block:*
  Example:
- *BEGIN*
- *SELECT ...*
- *SELECT ...*
- *SELECT ...*
- *.*
- *.*
- *.*
- *exception*
- *WHEN NO_DATA_FOUND THEN /* catches all 'no data found' errors */*
- *...*
- *WHEN ZERO_DIVIDE THEN /* different types of */*
- *WHEN value_error THEN /* errors handled in same block */*
- *...*

From above code we can conclude that exception handling

1. *Improves **readability** by letting us isolate error-handling routines and thus providing robustness.*
2. *Provides **reliability**, instead of checking for dIFferent types of errors at every point we can simply write them in exception block and IF error exists exception will be raised thus helping the programmer to find out the type of error and eventually resolve it.*

SQL Trigger | Student Database

Trigger: A trigger is a stored procedure in database which automatically invokes whenever a special event in the database occurs. For example, a trigger can be invoked when a row is inserted into a specified table or when certain table columns are being updated.

Syntax:

```
create trigger [trigger_name]

[before | after]

{insert | update | delete}

on [table_name]

[for each row]

[trigger_body]
```

Explanation of syntax:

1. create trigger [trigger_name]: Creates or replaces an existing trigger with the trigger_name.
2. [before | after]: This specifies when the trigger will be executed.
3. {insert | update | delete}: This specifies the DML operation.
4. on [table_name]: This specifies the name of the table associated with the trigger.
5. [for each row]: This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected.
6. [trigger_body]: This provides the operation to be performed as trigger is fired

BEFORE and AFTER of Trigger:

BEFORE triggers run the trigger action before the triggering statement is run. AFTER triggers run the trigger action after the triggering statement is run.

Example:

Given Student Report Database, in which student marks assessment is recorded. In such schema, create a trigger so that the total and percentage of specified marks is automatically inserted whenever a record is insert.

Here, as trigger will invoke before record is inserted so, BEFORE Tag can be used.

Suppose the database Schema –

```
mysql> desc Student;

+-------+-------------+------+-----+---------+----------------+

| Field | Type        | Null | Key | Default | Extra          |

+-------+-------------+------+-----+---------+----------------+

| tid   | int(4)      | NO   | PRI | NULL    | auto_increment |

| name  | varchar(30) | YES  |     | NULL    |                |

| subj1 | int(2)      | YES  |     | NULL    |                |

| subj2 | int(2)      | YES  |     | NULL    |                |
```

```
| subj3 | int(2)      | YES  |     | NULL    |                |
| total | int(3)      | YES  |     | NULL    |                |
| per   | int(3)      | YES  |     | NULL    |                |
+-------+-------------+------+-----+---------+----------------+
7 rows in set (0.00 sec)
```

SQL Trigger to problem statement.

```
create trigger stud_marks
before INSERT
on
Student
for each row
set Student.total = Student.subj1 + Student.subj2 + Student.subj3,
Student.per = Student.total * 60 / 100;
```

Above SQL statement will create a trigger in the student database in which whenever subjects marks are entered, before inserting this data into the database, trigger will compute those two values and insert with the entered values. i.e.,

```
mysql> insert into Student values(0, "ABCDE", 20, 20, 20, 0, 0);
Query OK, 1 row affected (0.09 sec)


mysql> select * from Student;
+-----+-------+-------+-------+-------+-------+------+
| tid | name  | subj1 | subj2 | subj3 | total | per  |
+-----+-------+-------+-------+-------+-------+------+
| 100 | ABCDE |    20 |    20 |    20 |    60 |   36 |
+-----+-------+-------+-------+-------+-------+------+
1 row in set (0.00 sec)
```

In this way trigger can be creates and executed in the databases.