Unit @3

# 1). 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.

A schema that suffers from the “Spreadsheet Syndrome” is subject to data redundancies, data anomalies, and various inefficiencies. The cure for “Spreadsheet Syndrome” is database normalization.

Database normalization is a process by which an existing schema is modified to bring its component tables into compliance with a series of progressive normal forms. The concept of database normalization was first introduced by Edgar Frank Codd in his paper A Relational Model of Data for Large Shared Data Banks.

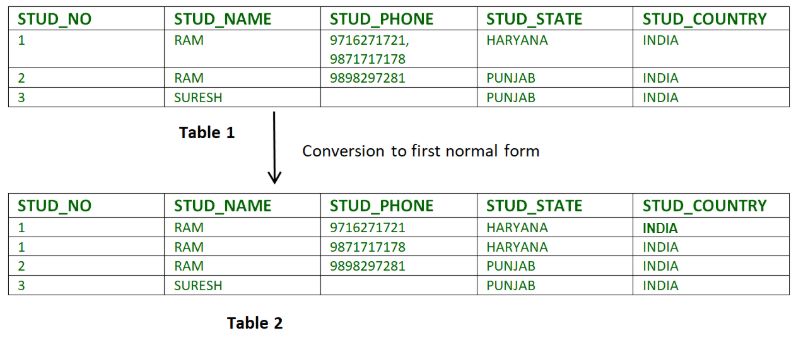
# 2). Normal Forms in DBMS

**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.

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.



### 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

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** **Table 2**

STUD\_NO COURSE\_NO COURSE\_NO COURSE\_FEE

1 C1 C1 1000

2 C2 C2 1500

1 C4 C3 1000

4 C3 C4 2000

4 C1 C5 2000

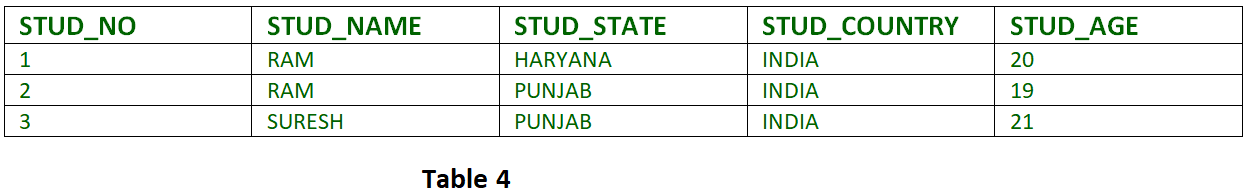
2 C5

**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.

### 3. Third Normal Form –

A relation is in third normal form, if there is **no transitive dependency** for non-prime attributes as well as it is in second normal form.  
A relation is in 3NF if **at least one of the following condition holds** in every non-trivial function dependency X –> Y

* 1. X is a super key.
  2. Y is a prime attribute (each element of Y is part of some candidate key).



### 4. Boyce-Codd Normal Form (BCNF) –

A relation R is in BCNF if R is in Third Normal Form and for every FD, LHS is super key. A relation is in BCNF iff in every non-trivial functional dependency X –> Y, X is a super key.

* + - **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.

# 3). Functional Dependency in DBMS:

**Functional Dependency (FD)** is a constraint that determines the relation of one attribute to another attribute in a Database Management System (DBMS). Functional Dependency helps to maintain the quality of data in the database. It plays a vital role to find the difference between good and bad database design.

A functional dependency is denoted by an arrow “→”. The functional dependency of X on Y is represented by X → Y. Let’s understand Functional Dependency in DBMS with example.

| **Employee number** | **Employee Name** | **Salary** | **City** |
| --- | --- | --- | --- |
| 1 | Dana | 50000 | San Francisco |
| 2 | Francis | 38000 | London |
| 3 | Andrew | 25000 | Tokyo |

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

Rules of Functional Dependencies

Below are the Three most important rules for Functional Dependency in Database:

Reflexive rule –. If X is a set of attributes and Y is\_subset\_of X, then X holds a value of Y.

Augmentation rule: When x -> y holds, and c is attribute set, then ac -> bc also holds. That is adding attributes which do not change the basic dependencies.

Transitivity rule: This rule is very much similar to the transitive rule in algebra if x -> y holds and y -> z holds, then x -> z also holds. X -> y is called as functionally that determines y.

Types of Functional Dependencies in DBMS

There are mainly four types of Functional Dependency in DBMS. Following are the types of Functional Dependencies in DBMS:

Multivalued Dependency

Trivial Functional Dependency

Non-Trivial Functional Dependency

Transitive Dependency

## 4). **What is Decomposition in DBMS?**

The term decomposition refers to the process in which we break down a table in a database into various elements or parts. Thus, decomposition replaces a given relation with a collection of various smaller relations. Thus, in a database, we can make any table break down into multiple tables when we want to collect a particular set of data.

Decomposition must always be lossless. This way, we can rest assured that the data/information that was there in the original relation can be reconstructed accurately on the basis of the decomposed relations. In case the relation is not decomposed properly, then it may eventually lead to problems such as information loss.

## **Types of Decomposition**

Decomposition is of two major types in DBMS:

* Lossless
* Lossy

### **1. Lossless Decomposition**

A decomposition is said to be lossless when it is feasible to reconstruct the original relation R using joins from the decomposed tables. It is the most preferred choice. This way, the information will not be lost from the relation when we decompose it. A lossless join would eventually result in the original relation that is very similar.

For example,

Let us take ‘A’ as the Relational Schema, having an instance of ‘a’. Consider that it is decomposed into: A1, A2, A3, . . . . An; with instance: a1, a2, a3, . . .. an, If a1 ⋈ a2 ⋈ a3 . . . . ⋈ an, then it is known as ‘Lossless Join Decomposition’. Read more about [Lossless Decomposition in DBMS](https://byjus.com/gate/lossless-decomposition-in-dbms/) here.

### **2. Lossy Decomposition**

Just like the name suggests, whenever we decompose a relation into multiple relational schemas, then the loss of data/information is unavoidable whenever we try to retrieve the original relation. Read more about [Lossy Decomposition in DBMS](https://byjus.com/gate/lossy-decomposition-in-dbms/) here.

## **Properties of Decomposition**

Decomposition must have the following properties:

1. Decomposition Must be Lossless

2. Dependency Preservation

3. Lack of Data Redundancy

### **1. Decomposition Must be Lossless**

Decomposition must always be lossless, which means the information must never get lost from a decomposed relation. This way, we get a guarantee that when joining the relations, the join would eventually lead to the same relation in the result as it was actually decomposed.

### **2. Dependency Preservation**

Dependency is a crucial constraint on a database, and a minimum of one decomposed table must satisfy every dependency. If {P → Q} holds, then two sets happen to be dependent functionally. Thus, it becomes more useful when checking the dependency if both of these are set in the very same relation. This property of decomposition can be done only when we maintain the functional dependency. Added to this, this property allows us to check various updates without having to compute the database structure’s natural join.

### **3. Lack of Data Redundancy**

It is also commonly termed as a repetition of data/information. According to this property, decomposition must not suffer from data redundancy. When decomposition is careless, it may cause issues with the overall data in the database. When we perform normalization, we can easily achieve the property of lack of data redundancy.

### **What are the advantages of decomposition?**

Decomposition is a process that saves a lot of time. For instance, the code that is there for a complex program could be easily run to multiple lines of code. In case we make a mistake, it would take a very prolonged time to discover. Another advantage of decomposition is that it lets programmers copy useful chunks of code and then reuse them easily for various other programs.

6). 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 ⋈ R2 ⊃ R
* Decomposition is lossless if R1 ⋈ R2 = R

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

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)

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

Att(R1) ∩ Att(R2) ≠ Φ

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:

First condition holds true as Att(R1) U Att(R2) = (ABC) U (AD) = (ABCD) = Att(R).

Second condition holds true as Att(R1) ∩ Att(R2) = (ABC) ∩ (AD) ≠ Φ

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,

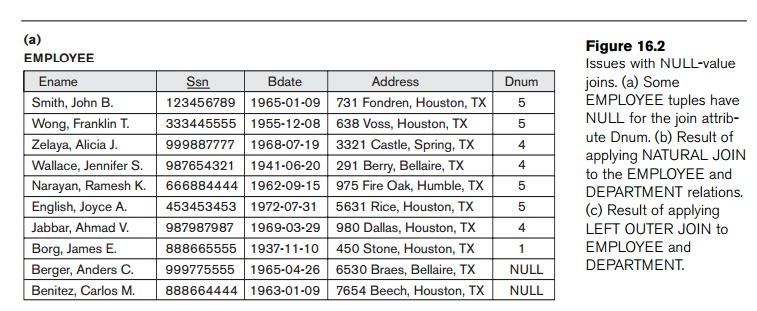
A->B can be ensured in R1(AB) and C->D can be ensured in R2(CD). Hence it is dependency preserving decomposition.  
So, the correct option is C.

7). **1. Problems with NULL Values and Dangling Tuples**

We must carefully consider the problems associated with NULLs when designing a relational database schema. There is no fully satisfactory relational design theory as yet that includes NULL values. One problem occurs when some tuples have NULL values for attributes that will be used to join individual relations in the decomposition. To illustrate this, consider the database shown in Figure 16.2(a), where two relations EMPLOYEE and DEPARTMENT are shown. The last two employee tuples— ‘Berger’ and ‘Benitez’—represent newly hired employees who have not yet been assigned to a department (assume that this does not violate any integrity constraints). Now suppose that we want to retrieve a list of (Ename, Dname) values for all the employees. If we apply the NATURAL JOIN operation on EMPLOYEE and DEPARTMENT (Figure 16.2(b)), the two aforementioned tuples will *not* appear in the result. The OUTER JOIN operation, discussed in Chapter 6, can deal with this problem. Recall that if we take the LEFT OUTER JOIN of EMPLOYEE with DEPARTMENT, tuples in EMPLOYEE that have NULL for the join attribute will still appear in the result, joined with an *imaginary* tuple in DEPARTMENT that has NULLs for all its attribute values. Figure 16.2(c) shows the result.

In general, whenever a relational database schema is designed in which two or more relations are interrelated via foreign keys, particular care must be devoted to watch-ing for potential NULL values in foreign keys. This can cause unexpected loss of information in queries that involve joins on that foreign key. Moreover, if NULLs occur in other attributes, such as Salary, their effect on built-in functions such as SUM and AVERAGE must be carefully evaluated.

A related problem is that of *dangling tuples*, which may occur if we carry a decomposition too far. Suppose that we decompose the EMPLOYEE relation in Figure 16.2(a) further into EMPLOYEE\_1 and EMPLOYEE\_2, shown in Figure 16.3(a) and 16.3(b). If we apply the NATURAL JOIN operation to EMPLOYEE\_1 and EMPLOYEE\_2, we get the original EMPLOYEE relation. However, we may use the alternative representation, shown in Figure 16.3(c), where we *do not include a tuple*



8).

Multivalued Dependency

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

**Example:** Suppose there is a bike manufacturer company which produces two colors(white and black) of each model every year.

|  |  |  |
| --- | --- | --- |
| **BIKE\_MODEL** | **MANUF\_YEAR** | **COLOR** |
| M2011 | 2008 | White |
| M2001 | 2008 | Black |
| M3001 | 2013 | White |
| M3001 | 2013 | Black |
| M4006 | 2017 | White |
| M4006 | 2017 | Black |

Here columns COLOR and MANUF\_YEAR are dependent on BIKE\_MODEL and independent of each other.

In this case, these two columns can be called as multivalued dependent on BIKE\_MODEL. The representation of these dependencies is shown below:

1. BIKE\_MODEL   →  →  MANUF\_YEAR
2. BIKE\_MODEL   →  →  COLOR

This can be read as "BIKE\_MODEL multidetermined MANUF\_YEAR" and "BIKE\_MODEL multidetermined COLOR".

# 9). What is Query Optimization?

Query optimization is of great importance for the performance of a relational database, especially for the execution of complex SQL statements. A query optimizer decides the best methods for implementing each query.

The query optimizer selects, for instance, whether or not to use indexes for a given query, and which join methods to use when joining multiple tables. These decisions have a tremendous effect on SQL performance, and query optimization is a key technology for every application, from operational Systems to data warehouse and analytical systems to content management systems.

There is the various principle of Query Optimization are as follows −

* **Understand how your database is executing your query** − The first phase of query optimization is understanding what the database is performing. Different databases have different commands for this. For example, in MySQL, one can use the “EXPLAIN [SQL Query]” keyword to see the query plan. In Oracle, one can use the “EXPLAIN PLAN FOR [SQL Query]” to see the query plan.
* **Retrieve as little data as possible** − The more information restored from the query, the more resources the database is required to expand to process and save these records. For example, if it can only require to fetch one column from a table, do not use ‘SELECT \*’.
* **Store intermediate results** − Sometimes logic for a query can be quite complex. It is possible to produce the desired outcomes through the use of subqueries, inline views, and UNION-type statements. For those methods, the transitional results are not saved in the database but are

directly used within the query. This can lead to achievement issues, particularly when the transitional results have a huge number of rows.

**There are broadly two ways a query can be optimized:**

1. Analyze and transform equivalent relational expressions: Try to minimize the tuple and column counts of the intermediate and final query processes (discussed here).
2. Using different algorithms for each operation: These underlying algorithms determine how tuples are accessed from the data structures they are stored in, indexing, hashing, data retrieval and hence influence the number of disk and block accesses (discussed in query processing).

### **Steps for Query Optimization**

Query optimization involves three steps, namely query tree generation, plan generation, and query plan code generation.

**Step 1 − Query Tree Generation**

A query tree is a tree data structure representing a relational algebra expression. The tables of the query are represented as leaf nodes. The relational algebra operations are represented as the internal nodes. The root represents the query as a whole.

During execution, an internal node is executed whenever its operand tables are available. The node is then replaced by the result table. This process continues for all internal nodes until the root node is executed and replaced by the result table.

For example, let us consider the following schemas −

EMPLOYEE

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| EmpID | EName | Salary | DeptNo | DateOfJoining |

DEPARTMENT

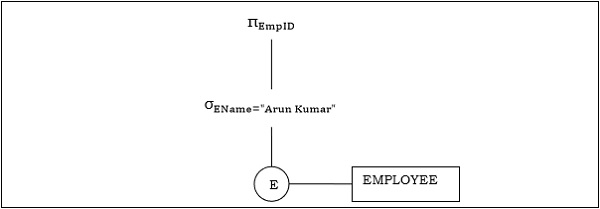
|  |  |  |
| --- | --- | --- |
| DNo | DName | Location |

### **Example 1**

Let us consider the query as the following.

$$\pi\_{EmpID} (\sigma\_{EName = \small "ArunKumar"} {(EMPLOYEE)})$$

The corresponding query tree will be −



**Step 2 − Query Plan Generation**

After the query tree is generated, a query plan is made. A query plan is an extended query tree that includes access paths for all operations in the query tree. Access paths specify how the relational operations in the tree should be performed. For example, a selection operation can have an access path that gives details about the use of B+ tree index for selection.

Besides, a query plan also states how the intermediate tables should be passed from one operator to the next, how temporary tables should be used and how operations should be pipelined/combined.

**Step 3− Code Generation**

Code generation is the final step in query optimization. It is the executable form of the query, whose form depends upon the type of the underlying operating system. Once the query code is generated, the Execution Manager runs it and produces the results.

9). Project and join operation of relational algbra.

**Project operation** selects (or chooses) certain attributes discarding other attributes. The Project operation is also known as vertical partitioning since it partitions the relation or table vertically discarding other columns or attributes.

**Notation:**

πA(R)

where ‘A’ is the attribute list, it is the desired set of attributes from the attributes of relation(R),  
symbol ‘π(pi)’  is used to denote the Project operator,  
R is generally a relational algebra expression, which results in a relation.

**Example –**

πAge(Student)

πDept, Sex(Emp)

**Example –**  
Given a relation Faculty (Class, Dept, Position) with the following tuples:

| Class | Dept | Position |
| --- | --- | --- |
| 5 | CSE | Assistant Professor |
| 5 | CSE | Assistant Professor |
| 6 | EE | Assistant Professor |
| 6 | EE | Assistant Professor |

**1. Project Class and Dept from Faculty –**

πClass, Dept(Faculty)

| Class | Dept |
| --- | --- |
| 5 | CSE |
| 6 | EE |

Here, we can observe that the degree (number of attributes) of resulting relation is 2, whereas the degree of Faculty relation is 3, So from this we can conclude that we may get a relation with varying degree on applying Project operation on a relation.

Hence, the degree of resulting relation is equal to the number of attribute in the attribute list ‘A’.

**2. Project Position from Faculty –**

πPosition(Faculty)

| Position |
| --- |
| Assistant Professor |

Here, we can observe that all the duplicate tuples are removed from the relation in the resulting relation. This is called as Duplicate elimination.

**3. Project Class from Faculty –**

πClass(Faculty)

| Class |
| --- |
| 5 |
| 6 |

**Important points:**

1. The Project operation removes duplicate tuples.
2. The Project operation is not commutative, that is :

πAttribute List 1(πAttribute List2(R)) != πAttribute List 2 (πAttribute List1(R))

1. The following expression is valid only if Attribute List 1 is a subset of Attribute List 2.

πAttribute List 1(πAttribute List2(R))

Moreover, writing the above expression is as good as writing the expression below:

πAttribute List 1(πAttribute List2(R)) = πAttribute List 1 (R)

1. The cardinality (number of tuples)  of resulting relation from a Project operation is:

1 <= πA(R) <= |R|

1. The degree (number of attributes) of resulting relation from a Project operation is equal to the number of attribute in the attribute list ‘A’.
2. In SQL, SELECT DISTINCT query is exactly as same as PROJECT here.

JOIN:\_

Join operation combines the relation R1 and R2 with respect to a condition. It is denoted by ⋈.

The different types of join operation are as follows −

* Theta join
* Natural join
* Outer join − It is further classified into following types −
  + Left outer join.
  + Right outer join.
  + Full outer join.

## **Theta join**

If we join R1 and R2 other than the equal to condition then it is called theta join/ non-equi join.

### **Example**

Consider R1 table

| **RegNo** | **Branch** | **Section** |
| --- | --- | --- |
| 1 | CSE | A |
| 2 | ECE | B |
| 3 | CIVIL | A |
| 4 | IT | B |
| 5 | IT | A |

### **Table R2**

| **Name** | **RegNo** |
| --- | --- |
| Bhanu | 2 |
| Priya | 4 |

R1 ⋈ R2 with condition R1.regno > R2.regno

| **RegNo** | **Branch** | **Section** | **Name** | **Regno** |
| --- | --- | --- | --- | --- |
| 3 | CIVIL | A | Bhanu | 2 |
| 4 | IT | B | Bhanu | 2 |
| 5 | IT | A | Bhanu | 2 |
| 5 | IT | B | Priya | 4 |

In the join operation, we select those rows from the cartesian product where R1.regno>R2.regno.

Join operation = select operation + cartesian product operation

### **Natural join**

If we join R1 and R2 on equal condition then it is called natural join or equi join. Generally, join is referred to as natural join.

Natural join of R1 and R2 is −

{ we select those tuples from cartesian product where R1.regno=R2.regno}

### **R1 ⋈ R2**

| **Regno** | **Branch** | **Section** | **Name** |
| --- | --- | --- | --- |
| 2 | - | - | Bhanu |
| 4 | - | - | priya |

### **Outer join**

It is an extension of natural join to deal with missing values of relation.

Consider R1 and R2 shown below −

### **Table R1**

| **RegNo** | **Branch** | **Section** |
| --- | --- | --- |
| 1 | CSE | A |
| 2 | ECE | B |
| 3 | CIVIL | A |
| 4 | IT | B |
| 5 | IT | A |

### **Table R2**

| **Name** | **Regno** |
| --- | --- |
| Bhanu | 2 |
| Priya | 4 |
| Hari | 7 |

Outer join is of three types. These are explained below −

**Left outer join**

It is denoted by R1 ⋈ R2.

| **RegNo** | **Branch** | **Section** | **Name** | **Regno** |
| --- | --- | --- | --- | --- |
| 2 | - | - | Bhanu | 2 |
| 4 | - | - | Priya | 4 |
| 1 | - | - | NULL | NULL |
| 3 | - | - | NULL | NULL |
| 5 | - | - | NULL | NULL |

Here all the tuples of R1(left table) appear in output.

The mismatching values of R2 are filled with NULL.

Left outer join = natural join + mismatch / extra tuple of R1

**Right outer join**

It is denoted by R1 ⋈ R2

Here all the tuples of R2(right table) appear in output. The mismatching values of R1 are filled with NULL.

| **RegNo** | **Branch** | **Section** | **Name** | **Regno** |
| --- | --- | --- | --- | --- |
| 2 | - | - | Bhanu | 2 |
| 4 | - | - | Priya | 4 |
| NULL | NULL | NULL | Hari | 7 |

Right outer join = natural join+ mismatch/extra tuple of R2.

**Full outer join**

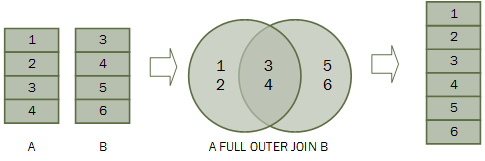
It is denoted by R1 ⋈ R2.

Full outer join=left outer join U right outer join.

| **RegNo** | **Branch** | **Section** | **Name** | **Regno** |
| --- | --- | --- | --- | --- |
| 2 | - | - | Bhanu | 2 |
| 4 | - | - | Priya | 4 |
| 1 | - | - | NULL | NULL |
| 3 | - | - | NULL | NULL |
| 5 | - | - | NULL | NULL |
| NULL | NULL | NULL | Hari | 7 |

### **Example**

Given below is the picture of full outer join −



# 10). What is heuristic optimization in DBMS?

## **Definition**

Heuristic designates a computational procedure that determines an optimal solution by iteratively trying to improve a candidate solution with regard to a given measure of quality. Heuristics make few or no assumptions about the problem being optimized and can search large spaces of candidate solutions toward finding optimal or near-optimal solutions at a reasonable computational cost without being able to guarantee either feasibility or optimality, or even in many cases to state how close to optimality a particular feasible solution is.

## **Rules**

Heuristic optimization transforms the expression-tree by using a set of rules which improve the performance. These rules are as follows −

* Perform the SELECTION process foremost in the query. This should be the first action for any SQL table. By doing so, we can decrease the number of records required in the query, rather than using all the tables during the query.
* Perform all the projection as soon as achievable in the query. Somewhat like a selection but this method helps in decreasing the number of columns in the query.
* Perform the most restrictive joins and selection operations. What this means is that select only those sets of tables and/or views which will result in a relatively lesser number of records and are extremely necessary in the query. Obviously any query will execute better when tables with few records are joined.

Some systems use only heuristics and the others combine heuristics with partial cost-based optimization.

## **Steps in heuristic optimization**

Let’s see the steps involve in heuristic optimization, which are explained below −

* Deconstruct the conjunctive selections into a sequence of single selection operations.
* Move the selection operations down the query tree for the earliest possible execution.
* First execute those selections and join operations which will produce smallest relations.
* Replace the cartesian product operation followed by selection operation with join operation.
* Deconstructive and move the tree down as far as possible.
* Identify those subtrees whose operations are pipelined.

# 12).Cost Based optimization

**Cost-Based Optimization:**  
For a given query and environment, the Optimizer allocates a cost in numerical form which is related to each step of a possible plan and then finds these values together to get a cost estimate for the plan or for the possible strategy. After calculating the costs of all possible plans, the Optimizer tries to choose a plan which will have the possible lowest cost estimate. For that reason, the Optimizer may be sometimes referred to as the Cost-Based Optimizer. Below are some of the features of the cost-based optimization-

1. The cost-based optimization is based on the cost of the query that to be optimized.
2. The query can use a lot of paths based on the value of indexes, available sorting methods, constraints, etc.
3. The aim of query optimization is to choose the most efficient path of implementing the query at the possible lowest minimum cost in the form of an algorithm.
4. The cost of executing the algorithm needs to be provided by the query Optimizer so that the most suitable query can be selected for an operation.
5. The cost of an algorithm also depends upon the cardinality of the input.

**Cost Estimation:**  
To estimate the cost of different available execution plans or the execution strategies the query tree is viewed and studied as a data structure that contains a series of basic operation which are linked in order to perform the query. The cost of the operations that are present in the query depends on the way in which the operation is selected such that, the proportion of select operation that forms the output. It is also important to know the expected cardinality of an operation output. The cardinality of the output is very important because it forms the input to the next operation.   
The cost of optimization of the query depends upon the following-

1. **Cardinality-**   
   Cardinality is known to be the number of rows that are returned by performing the operations specified by the query execution plan. The estimates of the cardinality must be correct as it highly affects all the possibilities of the execution plan.
2. **Selectivity-**  
   Selectivity refers to the number of rows that are selected. The selectivity of any row from the table or any table from the database almost depends upon the condition. The satisfaction of the condition takes us to the selectivity of that specific row. The condition that is to be satisfied can be any, depending upon the situation.
3. **Cost-**  
   Cost refers to the amount of money spent on the system to optimize the system. The measure of cost fully depends upon the work done or the number of resources used.

The first step is to use ANALYZE TABLE COMPUTE STATISTICS SQL command to compute table statistics. Use DESCRIBE EXTENDED SQL command to inspect the statistics.

**Table Statistics:**  
The table statistics can be computed for tables, partitions, and columns and are as follows-

1. **Total size**(in bytes) of a table or table partitions.
2. **Row count** of a table or table partitions.
3. **Column statistics**like min, max, num\_nulls, distinct\_count, avg\_col\_len, max\_col\_len, histogram.

**ANALYZE TABLE COMPUTE STATISTICS SQL Command:**  
Cost-Based Optimization uses the statistics stored in a meta store i.e. external catalog using ANALYZE TABLE SQL command-

ANALYZE TABLE tableIdentifier partitionSpec;

COMPUTE STATISTICS (NOSCAN | FOR COLUMNS identifierSeq);

Depending on the variant, ANALYZE TABLE computes different statistics, i.e. of a table, partitions, or columns-

* ANALYZE TABLE with neither PARTITION specification nor FOR COLUMNS clause.
* ANALYZE TABLE with PARTITION specification (but no FOR COLUMNS clause).
* ANALYZE TABLE with FOR COLUMNS clause (but no PARTITION specification).

**DESCRIBE EXTENDED SQL Command:**  
The statistics of a table can be viewed, partitions, or a column (stored in a meta store) using DESCRIBE EXTENDED SQL command-

(DESC | DESCRIBE) TABLE? (EXTENDED | FORMATTED);

tableIdentifier partitionSpec? describeColName;

**Cost Components Of Query Execution:**  
The following are the cost components of the execution of a query-

1. **Access cost to secondary storage-**   
   This can be the cost of searching, reading, or writing data blocks that originally found on the secondary storage, especially on the disk. The cost of searching for records in a file also depends upon the type of access structure that file has.
2. **Memory usage cost-**  
   The cost of memory usage can be calculated simply by using the number of memory buffers that are needed for the execution of the query.
3. **Storage cost-**  
   The storage cost is the cost of storing any intermediate files(files that are the result of processing the input but are not exactly the result) that are generated by the execution strategy for the query.
4. **Computational cost-**  
   This is the cost of performing the memory operations that are available on the record within the data buffers. Operations like searching for records, merging records, or sorting records. This can also be called the CPU cost.
5. **Communication cost-**  
   This is the cost that is associated with sending or communicating the query and its results from one place to another. It also includes the cost of transferring the table and results to the various sites during the process of query evaluation.