**Unit-4 notes**

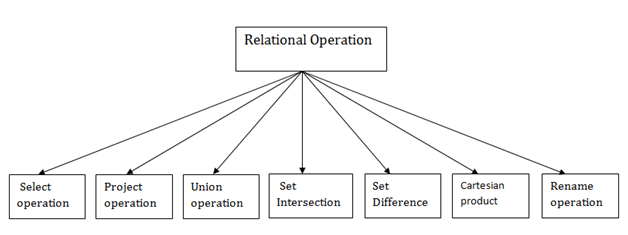
**Relational algebra:**

Relational algebra is a procedural query language, which takes instances of relations as input and yields instances of relations as output. It uses operators to perform queries. An operator can be either unary or binary. They accept relations as their input and yield relations as their output. Relational algebra is performed recursively on a relation and intermediate results are also considered relations.

The relational algebra is a theoretical procedural query language which takes an instance of relations and does operations that work on one or more relations to describe another relation without altering the original relation(s). Thus, both the operands and the outputs are relations. So the output from one operation can turn into the input to another operation, which allows expressions to be nested in the relational algebra, just as you nest arithmetic operations. This property is called closure: relations are closed under the algebra, just as numbers are closed under arithmetic operations.

The relational algebra is a relation-at-a-time (or set) language where all tuples are controlled in one statement without the use of a loop. There are several variations of syntax for relational algebra commands, and you use a common symbolic notation for the commands and present it informally.

**Types of Relational operation:**



**Select Operation (σ)**

It selects tuples that satisfy the given predicate from a relation.

**Notation** − σ*p*(r)

Where σ stands for selection predicate and r stands for relation. *p* is prepositional logic formula which may use connectors like and, or, and not. These terms may use relational operators like − =, ≠, ≥, < ,  >,  ≤.

For example −

σ*subject = "database"*(Books)

**Output** − Selects tuples from books where subject is 'database'.

σsubject = "database" and price = "450"(Books)

**Output** − Selects tuples from books where subject is 'database' and 'price' is 450.

σsubject = "database" and price = "450" or year > "2010"(Books)

**Output** − Selects tuples from books where subject is 'database' and 'price' is 450 or those books published after 2010.

## Project Operation (∏)

It projects column(s) that satisfy a given predicate.

Notation − ∏A1, A2, An (r)

Where A1, A2 , An are attribute names of relation **r**.

Duplicate rows are automatically eliminated, as relation is a set.

**For example** −

∏subject, author (Books)

Selects and projects columns named as subject and author from the relation Books.

## Union Operation (∪)

It performs binary union between two given relations and is defined as −

r ∪ s = { t | t ∈ r or t ∈ s}

**Notation** − r U s

Where **r** and **s** are either database relations or relation result set (temporary relation).

For a union operation to be valid, the following conditions must hold −

* **r**, and **s** must have the same number of attributes.
* Attribute domains must be compatible.
* Duplicate tuples are automatically eliminated.

∏ author (Books) ∪ ∏ author (Articles)

**Output** − Projects the names of the authors who have either written a book or an article or both.

### **Set Intersection**

Suppose there are two tuples R and S. The set intersection operation contains all tuples that are in both R & S.It is denoted by intersection ∩.

Notation: R ∩ S

**Example:** Using the above DEPOSITOR table and BORROW table

∏ CUSTOMER\_NAME (BORROW) ∩ ∏ CUSTOMER\_NAME (DEPOSITOR)

## Set Difference (−)

The result of set difference operation is tuples, which are present in one relation but are not in the second relation.

**Notation** − **r** − **s**

Finds all the tuples that are present in **r** but not in **s**.

∏ author (Books) − ∏ author (Articles)

**Output** − Provides the name of authors who have written books but not articles.

## Cartesian Product (Χ)

Combines information of two different relations into one.

**Notation** − r Χ s

Where **r** and **s** are relations and their output will be defined as −

r Χ s = { q t | q ∈ r and t ∈ s}

σauthor = 'navathe'(Books Χ Articles)

**Output** − Yields a relation, which shows all the books and articles written by navathe.

## Rename Operation (ρ)

The results of relational algebra are also relations but without any name. The rename operation allows us to rename the output relation. 'rename' operation is denoted with small Greek letter **rho***ρ*.

**Notation** − *ρ*x (E)

Where the result of expression **E** is saved with name of **x**.

**Relational calculus:**

Relational calculus in RDBM is referring to the non-procedural query language that emphasizes on the concept of what to for the data management rather how to do those. The relational calculus provides descriptive information about the queries to achieve the required result by using mathematical predicates calculus notations. It is an integral part of the relational data model. The relational calculus in DBMS uses specific terms such as tuple and domain to describe the queries. Some of the other related common terminologies for relational calculus are variables, constant, Comparison operators, logical connectives, and quantifiers. It creates the expressions that are also known as formulas with unbound formal variables.

**Types of Relational Calculus:**

Tuple and domain are the major components of relational calculus. A result tuple is an assignment of constants to these.

Variables that make the formula evaluate to be true. There are two types of relational calculus available in DBMS.

* Tuple relational calculus (TRC)
* Domain relational calculus (DRC)

Both the types of relational calculus are semantically similar for operating in DBMS data retrieval definitions.

**TRC:**

Tuple relational calculus works on filtering the tuples based on the specified conditions.TRC is the variable range over the tuples and is a type of simple subset of the first-order logic.TRC considers tuples as equal status as variables, and field referencing can be used to select the tuple parts. It is represented using letter ‘T’ and conditions with the pipe symbol and enclosing curly braces.

Syntax of TRC:

{T | Conditions)

The TRC syntax supports to denote the Table names or relation names, defining the tuple variables, and the column names. It uses the ‘.’  operator symbol to specify the column names with the table name.

TRC specifies the relation names with the Tuple variable name such as ’T’. Syntax of Relation definition in TRC:Relation(T)

For example, if the Product is the relation name, it can be denoted as Product(T). Similarly, TRC has the provision to specify the conditions. The condition is applicable for a particular attribute or the column.

For instance, if the data need to be represented for the particular product id of value 10, it can be denoted as T.product\_id=10, where T is the tuple variable that represents the row of the table.

Let us assume the Product table in the database as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Product\_id** | **Product Category** | **Product Name** | **Product Unit Price** |
| 8 | New | TV Unit 1 | $100 |
| 10 | New | TV Unit 2 | $120 |
| 12 | Existing | TV Cabinet | $77 |

Now to represent the relational calculus to return the product name that has the product id value as 10 from the product table, it can be denoted as with the tuple variable T.

T.Product Name | Product(T) AND T.Product\_id = 10

This relational calculus predicate describes what to do for getting the resultant tuple from the database.The result of the tuple relational calculus for the Product table will be:

|  |  |
| --- | --- |
| **Product\_id** | **Product Name** |
| 10 | TV Unit 2 |

#### **DRC**

The domain regional calculus works based on the filtering of the domain and the related attributes.DRC is the variable range over the domain elements or the filed values. It is a type of simple subset of first-order logic. It is domain-dependent compared to TRC is tuple dependent. In DRC the formal variables are explicit for the relational calculus representations. The domain attributes in DRC can be represented as C1, C2,…, Cn and the condition related to the attributes can be denoted as the formula defining the condition for fetching the F(C1, C2, …Cn )

Syntax of DRC in DBMS

{c1, c2,...,cn| F(c1, c2,... ,cn)}

Let us assume the  same Product table in the database as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Product\_id** | **Product Category** | **Product Name** | **Product Unit Price** |
| 8 | New | TV Unit 1 | $100 |
| 10 | New | TV Unit 2 | $120 |
| 12 | Existing | TV Cabinet | $77 |

DRC for  the  product name attribute from the Product table needs where the product id is 10, It will be demoted as:

{< Product Name, Product\_id> | ∈ Product ∧Product\_id> 10}

The result of the domain relational calculus for the Product table will be

|  |  |
| --- | --- |
| **Product\_id** | **Product Name** |
| 10 | TV Unit 2 |

Some of the commonly used logical operator notations for DRC are ∧ for AND,∨ for OR, and ┓ for NOT. similarly, the mathematical symbol **∈**refers to the relation “is an element of” or known as the set membership.

**Query Processing in DBMS:**

Query Processing is the activity performed in extracting data from the database. In query processing, it takes various steps for fetching the data from the database. The steps involved are:

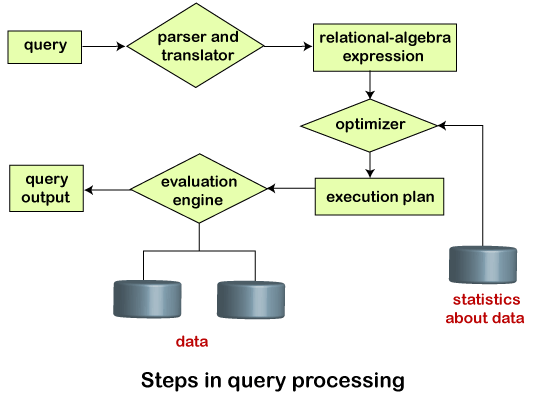
1. Parsing and translation
2. Optimization
3. Evaluation

The query processing works in the following way:

**Parsing and Translation**

As query processing includes certain activities for data retrieval. Initially, the given user queries get translated in high-level database languages such as SQL. It gets translated into expressions that can be further used at the physical level of the file system. After this, the actual evaluation of the queries and a variety of query -optimizing transformations and takes place. Thus before processing a query, a computer system needs to translate the query into a human-readable and understandable language. Consequently, SQL or Structured Query Language is the best suitable choice for humans. But, it is not perfectly suitable for the internal representation of the query to the system. Relational algebra is well suited for the internal representation of a query. The translation process in query processing is similar to the parser of a query. When a user executes any query, for generating the internal form of the query, the parser in the system checks the syntax of the query, verifies the name of the relation in the database, the tuple, and finally the required attribute value. The parser creates a tree of the query, known as 'parse-tree.' Further, translate it into the form of relational algebra. With this, it evenly replaces all the use of the views when used in the query.

Thus, we can understand the working of a query processing in the below-described diagram:



Suppose a user executes a query. As we have learned that there are various methods of extracting the data from the database. In SQL, a user wants to fetch the records of the employees whose salary is greater than or equal to 10000. For doing this, the following query is undertaken:

select emp\_name from Employee where salary>10000;

Thus, to make the system understand the user query, it needs to be translated in the form of relational algebra. We can bring this query in the relational algebra form as:

σsalary>10000 (πsalary (Employee))

πsalary (σsalary>10000 (Employee))

After translating the given query, we can execute each relational algebra operation by using different algorithms. So, in this way, a query processing begins its working.

**Evaluation**

For this, with addition to the relational algebra translation, it is required to annotate the translated relational algebra expression with the instructions used for specifying and evaluating each operation. Thus, after translating the user query, the system executes a query evaluation plan.

**Query Evaluation Plan:**

* In order to fully evaluate a query, the system needs to construct a query evaluation plan.
* The annotations in the evaluation plan may refer to the algorithms to be used for the particular index or the specific operations.
* Such relational algebra with annotations is referred to as Evaluation Primitives. The evaluation primitives carry the instructions needed for the evaluation of the operation.
* Thus, a query evaluation plan defines a sequence of primitive operations used for evaluating a query. The query evaluation plan is also referred to as the query execution plan.
* A query execution engine is responsible for generating the output of the given query. It takes the query execution plan, executes it, and finally makes the output for the user query.

**Optimization**

* The cost of the query evaluation can vary for different types of queries. Although the system is responsible for constructing the evaluation plan, the user does need not to write their query efficiently.
* Usually, a database system generates an efficient query evaluation plan, which minimizes its cost. This type of task performed by the database system and is known as Query Optimization.
* For optimizing a query, the query optimizer should have an estimated cost analysis of each operation. It is because the overall operation cost depends on the memory allocations to several operations, execution costs, and so on.

Finally, after selecting an evaluation plan, the system evaluates the query and produces the output of the query.

**Pitfalls in Relational database:**

Relational databases are widely used in many industries to store financial records, keep track of inventory and to keep records on employees. In a relational database, information is stored in tables (often called relations) which help organize and structure data. Even though they are widely used, relational databases have some drawbacks.

**Cost**

One disadvantage of relational databases is the expensive of setting up and maintaining the database system. In order to set up a relational database, you generally need to purchase special software. If you are not a programmer, you can use any number of products to set up a relational database. It does take time to enter in all the information and set up the program. If your company is large and you need a more robust database, you will need to hire a programmer to create a relational database using Structured Query Language (SQL) and a database administrator to maintain the database once it is built. Regardless of what data you use, you will have to either import it from other data like text files or Excel spreadsheets, or have the data entered at the keyboard. No matter the size of your company, if you store legally confidential or protected information in your database such as health information, social security numbers or credit card numbers, you will also have to secure your data against unauthorized access in order to meet regulatory standards.

**Abundance of Information**

Advances in the complexity of information cause another drawback to relational databases. Relational databases are made for organizing data by common characteristics. Complex images, numbers, designs and multimedia products defy easy categorization leading the way for a new type of database called object-relational database management systems. These systems are designed to handle the more complex applications and have the ability to be scalable.

**Stuctured Limits**

Some relational databases have limits on field lengths. When you design the database, you have to specify the amount of data you can fit into a field. Some names or search queries are shorter than the actual, and this can lead to data loss.

**Isolated Databases**

Complex relational database systems can lead to these databases becoming "islands of information" where the information cannot be shared easily from one large system to another. Often, with big firms or institutions, you find relational databases grew in separate divisions differently. For example, maybe the hospital billing department used one database while the hospital personnel department used a different database. Getting those databases to "talk" to each other can be a large, and expensive, undertaking, yet in a complex hospital system, all the databases need to be involved for good patient and employee care.

**Decomposition:**

* Decomposition is the process of breaking down in parts or elements.
* It replaces a relation with a collection of smaller relations.
* It breaks the table into multiple tables in a database.
* It should always be lossless, because it confirms that the information in the original relation can be accurately reconstructed based on the decomposed relations.
* If there is no proper decomposition of the relation, then it may lead to problems like loss of information.

## ****Types of Decomposition-****

 Decomposition of a relation can be completed in the following two ways-



## ****1. Lossless Join Decomposition-****

* Consider there is a relation R which is decomposed into sub relations R1 , R2 , …. , Rn.
* This decomposition is called lossless join decomposition when the join of the sub relations results in the same relation R that was decomposed.
* For lossless join decomposition, we always have-

**R1⋈ R2⋈ R3 ……. ⋈ Rn = R,where ⋈ is a natural join operator**

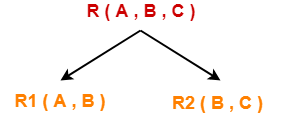
**Example-**

Consider the following relation R( A , B , C )-

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **C** |
| 1 | 2 | 1 |
| 2 | 5 | 3 |
| 3 | 3 | 3 |

**R( A , B , C )**

Consider this relation is decomposed into two sub relations R1( A , B ) and R2( B , C )-



The two sub relations are-

|  |  |
| --- | --- |
| **A** | **B** |
| 1 | 2 |
| 2 | 5 |
| 3 | 3 |

**R1( A , B )**

|  |  |
| --- | --- |
| **B** | **C** |
| 2 | 1 |
| 5 | 3 |
| 3 | 3 |

**R2( B , C )**

Now, let us check whether this decomposition is lossless or not.

For lossless decomposition, we must have-

R1 ⋈ R2 = R

Now, if we perform the natural join ( ⋈ ) of the sub relations R1 and R2 ,we get-

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **C** |
| 1 | 2 | 1 |
| 2 | 5 | 3 |
| 3 | 3 | 3 |

 This relation is same as the original relation R.

Thus, we conclude that the above decomposition is lossless join decomposition.

**NOTE-**

* Lossless join decomposition is also known as **non-additive join decomposition.**
* This is because the resultant relation after joining the sub relations is same as the decomposed relation.
* No extraneous tuples appear after joining of the sub-relations.

**2. Lossy Join Decomposition-**

Consider there is a relation R which is decomposed into sub relations R1 , R2 , …. , Rn.

This decomposition is called lossy join decomposition when the join of the sub relations does not result in the same relation R that was decomposed.

The natural join of the sub relations is always found to have some extraneous tuples.

For lossy join decomposition, we always have-

R1 ⋈ R2 ⋈ R3 ……. ⋈ Rn ⊃R ,where⋈ is a natural join operator

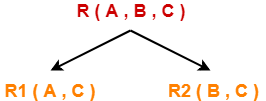
Example-

 Consider the following relation R( A , B , C )-

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **C** |
| 1 | 2 | 1 |
| 2 | 5 | 3 |
| 3 | 3 | 3 |

**R( A , B , C )**

 Consider this relation is decomposed into two sub relations as R1( A , C ) and R2( B , C )-



The two sub relations are-

|  |  |
| --- | --- |
| **A** | **C** |
| 1 | 1 |
| 2 | 3 |
| 3 | 3 |

**R1( A , B )**

|  |  |
| --- | --- |
| **B** | **C** |
| 2 | 1 |
| 5 | 3 |
| 3 | 3 |

**R2( B , C )**

Now, let us check whether this decomposition is lossy or not.

For lossy decomposition, we must have-

**R1⋈ R2⊃ R**

Now, if we perform the natural join ( ⋈ ) of the sub relations R1 and R2 we get-

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **C** |
| 1 | 2 | 1 |
| 2 | 5 | 3 |
| 2 | 3 | 3 |
| 3 | 5 | 3 |
| 3 | 3 | 3 |

This relation is not same as the original relation R and contains some extraneous tuples.

Clearly, R1⋈ R2 ⊃ R.

Thus, we conclude that the above decomposition is lossy join decomposition.

NOTE-

* Lossy join decomposition is also known as careless decomposition.
* This is because extraneous tuples get introduced in the natural join of the sub-relations.
* Extraneous tuples make the identification of the original tuples difficult.

**Dependency Preserving:**

* It is an important constraint of the database.
* In the dependency preservation, at least one decomposed table must satisfy every dependency.
* If a relation R is decomposed into relation R1 and R2, then the dependencies of R either must be a part of R1 or R2 or must be derivable from the combination of functional dependencies of R1 and R2.
* For example, suppose there is a relation R (A, B, C, D) with functional dependency set (A->BC). The relational R is decomposed into R1(ABC) and R2(AD) which is dependency preserving because FD A->BC is a part of relation R1(ABC).

**Dependency preserving decomposition:**

A Dependency preserving decomposition of a relation R is R1, R2, R3...Rn concerning the set of Functional Dependencies FD if,

(FD1 ∪ FD2 ∪ ... ∪FDn)+ = FD+

where,

* FD1, FD2, FD3…...FDn Sets of Functional dependencies of relations R1, R2, R3 ...Rn.
* (FD1 U FD2 U FD3 U … U FDn)+ -> Closure of Union of all sets of functional dependencies.
* FD+ -> Closure of set of functional dependency FD of R.

With FD (FD1) R is decomposed or divided into R1 and with FD(FD2) into R2, then the possibility of three cases arise,

* FD1 ∪ FD2 = FD -> Decomposition is dependency preserving.
* FD1 ∪ FD2 is a subset of FD -> Not Dependency preserving.
* FD1 ∪ FD2 is a superset of FD -> This case is not possible.

For the lossless dependency preserving decomposition, the closure of the set of functional dependencies of discrete relations R1, R2, R3 ...Rn should be equal to the set of functional dependencies of the main relation R before decomposition.

Dependency preservation and Normalization process, both concepts works on some similarity. As in Normalization process, to change the form of a relationship into a higher normal form, the solution is by decomposing the relation into two or more relations, which is done by using the set of functional dependencies associated in the lower normal form state.

Problem: Let a relation R (A, B, C, D ) and functional dependency {AB –> C, C –> D, D –> A}. Relation R is decomposed into R1( A, B, C) and R2(C, D). Check whether decomposition is dependency preserving or not.

Solution:

R1(A, B, C) and R2(C, D)

Let us find closure of F1 and F2

To find closure of F1, consider all combination of

ABC. i.e., find closure of A, B, C, AB, BC and AC

Note ABC is not considered as it is always ABC

closure(A) = { A } // Trivial

closure(B) = { B } // Trivial

closure(C) = {C, A, D} but D can't be in closure as D is not present R1.

= {C, A}

C--> A // Removing C from right side as it is trivial attribute

closure(AB) = {A, B, C, D}

= {A, B, C}

AB -->C // Removing AB from right side as these are trivial attributes

closure(BC) = {B, C, D, A}

= {A, B, C}

BC -->A // Removing BC from right side as these are trivial attributes

closure(AC) = {A, C, D}

AC -->D // Removing AC from right side as these are trivial attributes

F1 {C--> A, AB --> C, BC --> A}.

Similarly F2 { C--> D }

In the original Relation Dependency { AB --> C , C --> D , D --> A}.

AB --> C is present in F1.

C --> D is present in F2.

D --> A is not preserved.

F1 U F2 is a subset of F. So given decomposition is not dependency preserving.

**Functional Dependency:**

The term functional dependency means the association between any two attributes. Typically, this relationship is demonstrated between the primary key and non-key attributes within the table of a Database Management System, where the non-key attribute is functionally dependent on the primary key attribute to operate. A Functional Dependency case in a table is termed as ‘Minimal’ if the non- key attribute has dependencies on the primary key attribute with the functional characteristics such as there is only one non-key attribute in the table, any change made in the primary key attribute brings in changes to the non-key attribute as well, and if any alteration made on the functional dependencies will affect the table contents of the primary key.

Syntax:Functional Dependency on any given table can be explained as,

X → Y

Here, the left side of the arrow is identified as a Determinant, while the right side of the arrow is identified as a Dependent. X will always be the primary key attribute and Y will be any dependent non- key attribute from the same table as the primary key. This shows X primary key attribute is functionally dependent on the Y non-key attribute. In other words, If column X attribute of the table uniquely identifies the column Y attribute of the same table, then the functional dependency of the column Y on the column X is symbolized as X → Y.

**Types of Functional Dependency**

The Scenario for Functional Dependencies can be classified into the below four types:

**1. Multivalued Functional Dependency in DBMS**

Multivalued Functional Dependency takes place in the conditions when there is more than one independent attribute with multiple values in the same table. The Multivalued Dependency case is a complete limitation between two sets of attributes in the relationship of Functional Dependency. It requires that certain row values can be present as a functional dependency connection. This can be represented as,

X → Y

X → Z

X → A,

Where X, Y, Z, A are attributes of the same table, X being the primary key and Y, Z, A is non- key attributes. Here Y, Z, A are functionally dependent on X, and not dependent on each other.

For better understanding, let us consider the below table,

|  |  |  |  |
| --- | --- | --- | --- |
| **Student\_ID** | **Student\_Name** | **Dept** | **DOB** |
| S\_001 | Sname01 | Computer | Jan-01 |
| S\_002 | Sname02 | Maths | Mar-07 |
| S\_003 | Sname03 | English | Sept-11 |

In this example, Student\_Name, Dept & DOB are not dependent on each other but are dependent on Student\_ID. In terms of Functional Dependency, Student\_ID is the determinant, Student\_Name, Dept, DOB are the dependents. Student\_ID is the primary key here, while Student\_Name, Dept, and DOB are non-key columns. Hence the dependence can be symbolized as,

Student\_ID → Student\_NameStudent\_ID → Dept Student\_ID → DOB

**2. Trivial Functional Dependency in DBMS**

The Trivial Functional Dependency is a set of attributes or columns that are known a trivial if the on- key-dependent attribute is a subset of the determinant attribute, which is a primary key attribute. This Trivial Functional Dependency scenario occurs when the primary key is formed by two columns, and one of which is functionally dependent on the combined set.

X → Y, where is a trivial functional dependency, if Y is a subset of X. Let us consider the below table,

|  |  |  |  |
| --- | --- | --- | --- |
| **Student\_ID** | **Student\_Name** | **Dept** | **DOB** |
| S\_001 | Sname01 | Computer | Jan-01 |
| S\_002 | Sname02 | Maths | Mar-07 |
| S\_003 | Sname03 | English | Sept-11 |

Here, if the primary key is a combination of the columns Student\_ID and Student\_Name, then the Student\_Name column is in Trivial Functional Dependency relationship with the primary key set [Student\_ID, Student\_Name]. Any changes made in the Student\_Name column will have its effects on the primary key set [Student\_ID, Student\_Name], as the Student\_Name column is a subset of the primary key attribute set. For a Student ID, S\_001, the primary key combination will be [S\_001, Sname01]. If a change to the name is made as Sname001, then the primary key combination will change as [S\_001, Sname001], as the Student\_Name column is a subset of the primary key.

**3. Non-Trivial Functional Dependency in DBMS**

A Non Trivial Functional Dependency is a normal functional dependency, where the non-key attribute is functionally dependent on a primary key attribute, without the occurrence of trivial functional dependency.

X → Y, where is a non-trivial functional dependency, if and only if Y is not a subset of X. Let us consider the below table,

Here, if the primary key is the column Student\_ID, and the Student\_Name column is not a subset of Student\_ID, then the Student\_Name column is in a non Trivial Functional Dependency relationship with the primary key Student\_ID.

|  |  |  |  |
| --- | --- | --- | --- |
| **Student\_ID** | **Student\_Name** | **Dept** | **DOB** |
| S\_001 | Sname01 | Computer | Jan-01 |
| S\_002 | Sname02 | Maths | Mar-07 |
| S\_003 | Sname03 | English | Sept-11 |

**4. Transitive Functional Dependency in DBMS**

A Transitive Functional Dependency is a type of functional dependency which happens when the non- key attribute is indirectly formed by its functional dependencies on the primary key attributes. Either the value or the known factors can be the reason for this type of Functional Dependency occurrence. The Transitive Functional Dependency can only occur in a relation of three or more non-key attributes that are functionally dependent on the primary key attribute.

Let us consider the below table to understand this,

|  |  |  |  |
| --- | --- | --- | --- |
| **Student\_ID** | **Student\_Name** | **Dept** | **DOB** |
| S\_0101\_C | Sname01 | Computer\_C | 01-01-1999 |
| T\_0307\_M | Tname02 | Maths\_M | 03-07-1998 |
| U\_0711\_E | Uname03 | English\_E | 07-11-1997 |

In this table, the Student\_ID column is the primary key. The values in the Student\_ID column are formed by the combination of the first letter from the Student\_Name column, last code from the Dept column and date & month from the DOB column. If any change is made in any of these columns will reflect changes in the primary key column, that is, the Student\_ID column. Any new record inserted in this table will also have a Student\_ID value formed from the combination of the other three non-key columns.

**Advantages of Functional Dependency**

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

**Inference Rule (IR):**

The Functional dependency has 6 types of inference rule:

* Reflexive Rule (IR1):In the reflexive rule, if Y is a subset of X, then X determines Y.

If X ⊇ Y then X → Y

* Augmentation Rule (IR2):The augmentation is also called as a partial dependency. In augmentation, if X determines Y, then XZ determines YZ for any Z.

If X → Y then XZ → YZ

* Transitive Rule (IR3):In the transitive rule, if X determines Y and Y determine Z, then X must also determine Z.

If X → Y and Y → Z then X → Z

* Union Rule (IR4):Union rule says, if X determines Y and X determines Z, then X must also determine Y and Z.

If X → Y and X → Z then X → YZ

* Decomposition Rule (IR5):Decomposition rule is also known as project rule. It is the reverse of union rule. This Rule says, if X determines Y and Z, then X determines Y and X determines Z separately.

If X → YZ then X → Y and X → Z

* Pseudo transitive Rule (IR6):In Pseudo transitive Rule, if X determines Y and YZ determines W, then XZ determines W.

If X → Y and YZ → W then XZ → W

**Closures of a set of functional dependencies:**

A Closure is a set of FDs is a set of all possible FDs that can be derived from a given set of FDs. It is also referred as a Complete set of FDs. If F is used to donate the set of FDs for relation R, then a closure of a set of FDs implied by F is denoted by F+. Let's consider the set F of functional dependencies given below:

F = {A -> B, B -> C, C -> D}

from F, it is possible to derive following dependencies.

A -> A ...By using Rule-4, Self-Determination.

A -> B ...Already given in F.

A -> C ...By using rule-3, Transitivity.

A -> D ...By using rule-3, Transitivity.

Now, by applyiing Rule-6 Union, it is possible to derive A+ -> ABCD and it can be denoted using A -> ABCD. All such type of FDs derived from each FD of F form a closure of F. Steps to determine F+example:

* Determine each set of attributes X that appears as a left hand side of some FD in F.
* Determine the set X+ of all attributes that are dependent on X, as given in above example.
* In other words, X+ represents a set of attributes that are functionally determined by X based on F. And, X+ is called the Closure of X under F.
* All such sets of X+, in combine, Form a closure of F.

**Algorithm : Determining X+, the closure of X under F.**

Input : Let F be a set of FDs for relation R.

Steps:

1. X+ = X //initialize X+ to X

2. For each FD : Y -> Z in F Do

If Y ⊆ X+ Then //If Y is contained in X+

X+ = X+ ∪ Z //add Z to X+

End If

End For

3. Return X+ //Return closure of X

Output : Closure X+ of X under F

**Attribute Closure:**

Attribute closure of an attribute set can be defined as set of attributes which can be functionally determined from it.

How to find attribute closure of an attribute set?

* To find attribute closure of an attribute set:
* Add elements of attribute set to the result set.
* Recursively add elements to the result set which can be functionally determined from the elements of the result set.

**Closure Of Functional Dependency : Examples**

Example-1 : Consider the table student\_details having (Roll\_No, Name,Marks, Location) as the attributes and having two functional dependencies.

FD1 :Roll\_No🡪 Name, Marks

FD2 : Name 🡪 Marks, Location

Now, We will calculate the closure of all the attributes present in the relation using the three steps mentioned below.

Step-1 : Add attributes present on the LHS of the first functional dependency to the closure.

{Roll\_no}+ = {Roll\_No}

Step-2 : Add attributes present on the RHS of the original functional dependency to the closure.

{Roll\_no}+ = {Roll\_No, Marks}

Step-3 : Add the other possible attributes which can be derived using attributes present on the RHS of the closure. So Roll\_No attribute cannot functionally determine any attribute but Name attribute can determine other attributes such as Marks and Location using 2nd Functional Dependency(Name [icon name="long-arrow-right" class="" unprefixed\_class=""] Marks, Location).

Therefore, complete closure of Roll\_No will be :

{Roll\_no}+ = {Roll\_No, Marks, Name, Location}

Similarly, we can calculate closure for other attributes too i.e “Name”.

Step-1 : Add attributes present on the LHS of the functional dependency to the closure.

{Name}+ = {Name}

Step-2 : Add the attributes present on the RHS of the functional dependency to the closure.

{Name}+ = {Name, Marks, Location}

Step-3 : Since, we don’t have any functional dependency where “Marks or Location” attribute is functionally determining any other attribute , we cannot add more attributes to the closure. Hence complete closure of Name would be :

{Name}+ = {Name, Marks, Location}

NOTE : We don’t have any Functional dependency where marks and location can functionally determine any attribute. Hence, for those attributes we can only add the attributes themselves in their closures. Therefore,

{Marks}+ = {Marks}

and

{Location}+ = { Location}

Example-2 : Consider a relation R(A,B,C,D,E) having below mentioned functional dependencies.

FD1 : A 🡪 BC

FD2 : C 🡪B

FD3 : D 🡪 E

FD4 : E 🡪D

Now, we need to calculate the closure of attributes of the relation R. The closures will be:

{A}+ = {A, B, C}

{B}+ = {B}

{C}+ = {B, C}

{D}+ = {D, E}

{E}+ = {E}

**Closure Of Functional Dependency : Calculating Candidate Key**

“A Candidate Key of a relation is an attribute or set of attributes that can determine the whole relation or contains all the attributes in its closure."

Let’s try to understand how to calculate candidate keys.

**Example-1 :**Consider the relation R(A,B,C) with given functional dependencies :

FD1 : A 🡪 B

FD2 : B 🡪C

Now, calculating the closure of the attributes as :

{A}+ = {A, B, C}

{B}+ = {B, C}

{C}+ = {C}

Clearly, “A” is the candidate key as, its closure contains all the attributes present in the relation “R”.

**Example-2 :** Consider another relation R(A, B, C, D, E) having the Functional dependencies :

FD1 : A 🡪 BC

FD2 : C 🡪 B

FD3 : D 🡪 E

FD4 : E 🡪D

Now, calculating the closure of the attributes as :

{A}+ = {A, B, C}

{B}+ = {B}

{C}+ = {C, B}

{D}+ = {E, D}

{E}+ = {E, D}

In this case, a single attribute is unable to determine all the attribute on its own like in previous example. Here, we need to combine two or more attributes to determine the candidate keys.

{A, D}+ = {A, B, C, D, E}

{A, E}+ = {A, B, C, D, E}

Hence, "AD" and "AE" are the two possible keys of the given relation “R”. Any other combination other than these two would have acted as extraneous attributes.

NOTE : Any relation “R” can have either single or multiple candidate keys.

**Closure Of Functional Dependency : Key Definitions**

**Prime Attributes :** Attributes which are indispensable part of candidate keys. For example : “A, D, E” attributes are prime attributes in above example-2.

**Non-Prime Attributes :** Attributes other than prime attributes which does not take part in formation of candidate keys. For example.

**Extraneous Attributes :** Attributes which does not make any effect on removal from candidate key.

For example :Consider the relation R(A, B, C, D) with functional dependencies :

FD1 : A🡪 BC

FD2 : B 🡪C

FD3 : D 🡪 C.

Here, Candidate key can be “AD” only. Hence,

Prime Attributes : A, D.

Non-Prime Attributes : B, C

Extraneous Attributes : B, C(As if we add any of the to the candidate key, it will remain unaffected). Those attributes, which if removed does not affect closure of that set.

**Irreducible set of functional dependency(Canonical cover):**

In DBMS,

A canonical cover is a simplified and reduced version of the given set of functional dependencies.

Since it is a reduced version, it is also called as Irreducible set.

**Characteristics-**

* Canonical cover is free from all the extraneous functional dependencies.
* The closure of canonical cover is same as that of the given set of functional dependencies.
* Canonical cover is not unique and may be more than one for a given set of functional dependencies.

**Need-**

* Working with the set containing extraneous functional dependencies increases the computation time.
* Therefore, the given set is reduced by eliminating the useless functional dependencies.
* This reduces the computation time and working with the irreducible set becomes easier.

**Steps To Find Canonical Cover:**

**Step-01**:Write the given set of functional dependencies in such a way that each functional dependency contains exactly one attribute on its right side.

Example-The functional dependency X → YZ will be written as-

X → Y,X → Z

**Step-02:**

* Consider each functional dependency one by one from the set obtained in Step-01.
* Determine whether it is essential or non-essential.

To determine whether a functional dependency is essential or not, compute the closure of its left side-

* Once by considering that the particular functional dependency is present in the set
* Once by considering that the particular functional dependency is not present in the set

Then following two cases are possible-

Case-01: Results Come Out to be Same-

If results come out to be same,

* It means that the presence or absence of that functional dependency does not create any difference.
* Thus, it is non-essential.
* Eliminate that functional dependency from the set.

Case-01: Results Come Out to be Different-

If results come out to be different,

* It means that the presence or absence of that functional dependency creates a difference.
* Thus, it is essential.
* Do not eliminate that functional dependency from the set

Mark that functional dependency as essential

**Step-03:**

* Consider the newly obtained set of functional dependencies after performing Step-02.
* Check if there is any functional dependency that contains more than one attribute on its left side.

Then following two cases are possible-

Case-01: No-

* There exists no functional dependency containing more than one attribute on its left side.
* In this case, the set obtained in Step-02 is the canonical cover.

Case-01: Yes-

* There exists at least one functional dependency containing more than one attribute on its left side.
* In this case, consider all such functional dependencies one by one.
* Check if their left side can be reduced.

Use the following steps to perform a check-

* Consider a functional dependency.
* Compute the closure of all the possible subsets of the left side of that functional dependency.
* If any of the subsets produce the same closure result as produced by the entire left side, then replace the left side with that subset.
* After this step is complete, the set obtained is the canonical cover

**Example:**

The following functional dependencies hold true for the relational scheme R ( W , X , Y , Z ) –

* X → W
* WZ → XY
* Y → WXZ

Write the irreducible equivalent for this set of functional dependencies.

**Solution:**

Step-01: Write all the functional dependencies such that each contains exactly one attribute on its right side-

X → W

WZ → X , WZ → Y

Y → W , Y → X , Y → Z

**Step-02**:Check the essentiality of each functional dependency one by one.

For X → W:

Considering X → W, (X)+ = { X , W }

Ignoring X → W, (X)+ = { X }

Now, Clearly, the two results are different. Thus, we conclude that X → W is essential and can not be eliminated.

For WZ → X:

Considering WZ → X, (WZ)+ = { W , X , Y , Z }

Ignoring WZ → X, (WZ)+ = { W , X , Y , Z }

Now , Clearly, the two results are same. Thus, we conclude that WZ → X is non-essential and can be eliminated.

Eliminating WZ → X, our set of functional dependencies reduces to-

X → W , WZ → Y , Y → W , Y → X , Y → Z

Now, we will consider this reduced set in further checks.

For WZ → Y:

Considering WZ → Y, (WZ)+ = { W , X , Y , Z }

Ignoring WZ → Y, (WZ)+ = { W , Z }

Now , Clearly, the two results are different. Thus, we conclude that WZ → Y is essential and can not be eliminated.

For Y → W:

Considering Y → W, (Y)+ = { W , X , Y , Z }

Ignoring Y → W, (Y)+ = { W , X , Y , Z }

Now, Clearly, the two results are same. Thus, we conclude that Y → W is non-essential and can be eliminated.

Eliminating Y → W, our set of functional dependencies reduces to-

X → W

WZ → Y

Y → X

Y → Z

For Y → X:

Considering Y → X, (Y)+ = { W , X , Y , Z }

Ignoring Y → X, (Y)+ = { Y , Z }

Now,Clearly, the two results are different.Thus, we conclude that Y → X is essential and can not be eliminated.

For Y → Z:

Considering Y → Z, (Y)+ = { W , X , Y , Z }

Ignoring Y → Z, (Y)+ = { W , X , Y }

Now,Clearly, the two results are different.Thus, we conclude that Y → Z is essential and can not be eliminated.

From here, our essential functional dependencies are-

X → W,WZ → Y,Y → X,Y → Z

**Step-03:**

Consider the functional dependencies having more than one attribute on their left side.

Check if their left side can be reduced.

In our set, Only WZ → Y contains more than one attribute on its left side.

Considering WZ → Y, (WZ)+ = { W , X , Y , Z }

Now,Consider all the possible subsets of WZ.

Check if the closure result of any subset matches to the closure result of WZ.

(W)+ = { W }

(Z)+ = { Z }

Clearly, None of the subsets have the same closure result same as that of the entire left side.

Thus, we conclude that we can not write WZ → Y as W → Y or Z → Y.

Thus, set of functional dependencies obtained in step-02 is the canonical cover.

Finally, the canonical cover is-

X → W

WZ → Y

Y → X

Y → Z

**Normalization:**

In a database, a huge amount of data gets stored in multiple tables. There can be the possibility of redundancy to be present in the data. So, Normalization in DBMS can be defined as the process which eliminates the redundancy from the data and ensures data integrity. Also, the normalization of data helps in removing the insert, update and delete anomalies.

**How Does Normalization work in DBMS?**

The normalization in the DBMS can be defined as a technique to design the schema of a database and this is done by modifying the existing schema which also reduces the redundancy and dependency of the data. So with Normalization, the unwanted duplication in data is removed along with the anomalies. In insert anomaly, the values such as null are not allowed to be inserted for a column.

In update anomaly, the data cannot be updated correctly because the same values occur multiple times in a column and in delete anomaly the deletion of a record creates inconsistency as it gets deleted from more than one row. So the aim of normalization is to remove redundant data as well as storing only related data in the table. This decreases the database size and the data gets logically stored in the database.

**Anomalies in Database**

1) Update Anomalies: When several instances of the same data are scattered across the database without proper relationship/link, it could cause strange conditions where a few of the instances will get updated with new values whereas some of them will not. This leaves the database in an inconsistent state.

2) Deletion Anomalies: Incomplete deletion of a particular data section which leaves some residual instances. The database creator remains unaware of such unwanted data as it is present at a different location.

3) Insertion Anomalies: This occurs when an attempt to insert data into a non-existent record.

Paying attention to these anomalies can help to maintain a consistent database.

**Advantages Of Normalization**

Here we can see why normalization is an attractive prospect in RDBMS concepts.

1) A smaller database can be maintained as normalization eliminates the duplicate data. Overall size of the database is reduced as a result.

2) Better performance is ensured which can be linked to the above point. As databases become lesser in size, the passes through the data becomes faster and shorter thereby improving response time and speed.

3) Narrower tables are possible as normalized tables will be fine-tuned and will have lesser columns which allows for more data records per page.

4) Fewer indexes per table ensures faster maintenance tasks (index rebuilds).

5) Also realizes the option of joining only the tables that are needed.

**Disadvantages of normalization**

1) More tables to join as by spreading out data into more tables, the need to join table’s increases and the task becomes more tedious. The database becomes harder to realize as well.

2) Tables will contain codes rather than real data as the repeated data will be stored as lines of codes rather than the true data. Therefore, there is always a need to go to the lookup table.

3) Data model becomes extremely difficult to query against as the data model is optimized for applications, not for ad hoc querying. (Ad hoc query is a query that cannot be determined before the issuance of the query. It consists of an SQL that is constructed dynamically and is usually constructed by desktop friendly query tools.). Hence it is hard to model the database without knowing what the customer desires.

4) As the normal form type progresses, the performance becomes slower and slower.

5) Proper knowledge is required on the various normal forms to execute the normalization process efficiently. Careless use may lead to terrible design filled with major anomalies and data inconsistency

**Types of normal forms:**

The database normalization process is divided into following the normal form:

1. First Normal Form (1NF)
2. Second Normal Form (2NF)
3. Third Normal Form (3NF)
4. Boyce-Codd Normal Form (BCNF)
5. Fourth Normal Form (4NF)
6. Fifth Normal Form (5NF)

**First normal form(1NF):**

The table or relation is said to be in First Normal Form if it does not contain any multi-valued or composite attributes. So the table or relation should contain only single-valued attributes for fulfilling the condition for First Normal Form.

Let us take the example of the STUDENT table as below:

|  |  |  |
| --- | --- | --- |
| **Roll** | **Name** | **Subject** |
| 19 | Rajesh | Math, Science |
| 23 | Supriya | History, English |
| 32 | Zack | Geography |

The above table is not in First Normal Form as this contains the multi-valued attribute. The below table is transformed into the First Normal Form as it contains only atomic values.

|  |  |  |
| --- | --- | --- |
| **Roll** | **Name** | **Subject** |
| 19 | Rajesh | Math |
| 19 | Rajesh | Science |
| 23 | Supriya | History |
| 23 | Supriya | English |
| 32 | Zack | Geography |

**Second Normal Form(2NF)**

Second Normal Form (2NF) is based on the concept of full functional dependency. Second Normal Form applies to relations with composite keys, that is, relations with a primary key composed of two or more attributes. A relation with a single-attribute primary key is automatically in at least 2NF. A relation that is not in 2NF may suffer from the update anomalies.

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.

In other words,

A relation that is in First Normal Form and every non-primary-key attribute is fully functionally dependent on the primary key, then the relation is in Second Normal Form (2NF).

Note – If the proper subset of candidate key determines non-prime attribute, it is called partial dependency.

The normalization of 1NF relations to 2NF involves the removal of partial dependencies. If a partial dependency exists, we remove the partially dependent attribute(s) from the relation by placing them in a new relation along with a copy of their determinant.

Consider the examples given below.

Example-1:

Consider table 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 1Table 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 dont 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:

Consider the functional dependencies for the relation R(X, Y, E, F).

{XY->EF, E->F}

We thus find the closure of (XY) which is {X, Y, E, F}

Since its closure contains all the attributes in the relation thus XY is the candidate key.For each functional dependency, i.e., XY->EF:

It does not contain any partial dependency as the non prime attributes depend on the whole of candidate key.

E->F: It does not contain any partial dependency as here the non prime attributes depend on each other only.

**Advantages of Second Normal Form:**

Below are a few of the advantages of using the second normal form.

1. Redundant data is reduced more effectively.
2. Data is consistent in the database.
3. It improves the flexibility in designing a Database.
4. It also improves the overall organization of data in the database.
5. It also improves the security of the database.

**Difference between 1NF and 2NF :**

| S.NO. | 1NF | 2NF |
| --- | --- | --- |
| 1. | In order to be in 1NF any relation must be atomic and should not contain any composite or multi-valued attributes. | In order to be in 2NF any relation must be in 1NF and should not contain any partial dependency. |
| 2. | The identification of functional dependency is not necessary for first normal form. | The identification of functional dependency is necessary for second normal form. |
| 3. | First Normal form only deals with the schema of the table and it does not handle the update anomalies. | Second normal form handles the update anomalies. |
| 4. | A relation in 1NF may or may not be in 2NF. | A relation in 2NF is always in 1NF. |
| 5. | The primary key in case of first normal form can be a composite key. | The primary key in case of second normal form cannot be a composite key in case it arises any partial dependency. |
| 6. | The main goal of first normal form is to eliminate the redundant data within the table. | The main goal of second normal form is to actually ensure the data dependencies. |
| 7. | The first normal form is less stronger than the second normal form. | The second normal form is comparatively more strong than first normal form. |

**Third Normal Form (3NF):**

Third normal form is a form used to normalize the database design to avoid duplication of data. In a Relational Database Management System, a huge amount of data gets stored across multiple tables and the storing as well as the retrieval and manipulation of data becomes easier with the introduction of the concept of the key which establishes the relationship among the tables. As we store a huge amount of data in the tables, there might be cases where duplicate records may get stored in the table which will result in inconsistent data. Also, the redundancy in the data will lead to various issues such as insert, update and delete anomalies.

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).

In other words,

A relation that is in First and Second Normal Form and in which no non-primary-key attribute is transitively dependent on the primary key, then it is in Third Normal Form (3NF).

Note – If A->B and B->C are two FDs then A->C is called transitive dependency.

The normalization of 2NF relations to 3NF involves the removal of transitive dependencies. If a transitive dependency exists, we remove the transitively dependent attribute(s) from the relation by placing the attribute(s) in a new relation along with a copy of the determinant.

Example:

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 attribute are on right sides of all functional dependencies are prime.

Note –

Third Normal Form (3NF) is considered adequate for normal relational database design because most of the 3NF tables are free of insertion, update, and deletion anomalies. Moreover, 3NF always ensures functional dependency preserving and lossless.

**Advantages of Third Normal Form**

Below are the advantages of Third Normal Form:

* Normalization increases the data quality as the unwanted data is reduced from the database. Even though the redundancy of the Second Normal Form is less as compared to the First Normal Form, it is still possible to have update anomalies. For example, if one tuple is updated only while others remains unchanged, the inconsistency of data will be there in the database.
* The transitive dependency creates the update anomalies and they can be removed by the usage of the Third Normal Form.
* The Third Normal Form is also considered to be the ample requirement to build a database as the tables in the Third Normal Form are devoid of insert, update or delete anomalies.
* The Third Normal Form removes the redundancy effectively so the data becomes consistent as well as maintains the data integrity. As the redundancy is reduced, the database becomes less in size and also the duplication of data is reduced which also improves the performance.

**Difference between 2NF and 3NF :**

| S.NO. | 2NF(Second Normal Form) | 3NFhird Normal Form) |
| --- | --- | --- |
| 1. | It is already in 1NF. | It is already in 1NF as well as in 2NF also. |
| 2. | In 2NF non-prime attributes are allowed to be functionally dependent on non-prime attributes. | In 3NF non-prime attributes are only allowed to be functionally dependent on Super key of relation. |
| 3. | No partial functional dependency of non-prime attributes are on any proper subset of candidate key is allowed. | No transitive functional dependency of non-prime attributes on any super key is allowed. . |
| 4. | Stronger normal form than 1NF but lesser than 3NF | Stronger normal form than 1NF and 2NF. |
| 5. | It eliminates repeating groups in relation. | It virtually eliminates all the redundancies. |
| 6. | The goal of the second normal form is to eliminate redundant data. | The goal of the third normal form is to ensure referential integrity. |

**BCNF:**

BCNF can be expanded as Boyce Codd Normal Form, the fourth form of normalization on a database. It is necessary to normalize the Multidimensional database systems up to the last level of normalization until there is no more space for normalization to be carried out anymore. Normalization reduces or removes redundancy in the data, in turn, to maintain the database without any duplicate values and bring inconsistency to the data in the database. BCNF can be applied to a database that obeys two conditions; namely, it should be in 3NF stage and when at least one of the reference tables consists of a primary key.

A table is said to be BCNF when it satisfies the below two concepts-

1. It is in 3NF.
2. A->B, A should be a super key or a candidate key for any functional dependency. In other words, if B is a prime attribute, A cannot be a non-prime attribute.

**Decomposition into BCNF:**

When a table is in 3NF, it may or may not be in the Boyce Codd Normal Form. Each table/relation will have a set of functional dependency. If the FD does not satisfy the second condition of BCNF, the table is decomposed (breaking into smaller tables) recursively until all the functional dependency meets the super key criteria.

The algorithm to be followed for decomposition is,

* Determine the functional dependency that violates the BCNF.
* For every functional dependency X->Y which violates, decompose the relation into R-Y and XY. Here R is a relation.
* Repeat until all the relations satisfy BCNF.

**Example#1:**

Let’s consider a Relation R with five attributes.

R=ABCDE

The functional dependencies are

FD = {A -> BC, C -> DE)

Candidate keys are {A}

Algorithm:

Inspect each of the FD to check whether it satisfies the second condition of BCNF as it is in 3NF.

The first FD A -> BC since A is a key for R this FD does not violate BCNF.

Second FD C -> DE, C is not a key of R. We decompose R into (CDE) (ABC).

The two schemas are created with the FD attributes, which violates and the other with original attributes minus the right-hand side of the violating FD. Now we will check both the newly created relations to check whether they are in BCNF or not. A is the key in (ABC), and C is the key (CDE) they do not violate BCNF. Thus the relation is in BCNF.

**Example #2**

Let’s consider a Relation R with five attributes.

R = (WXYZ)

The functional dependencies are

F = {WX -> Y, X -> Z; Y -> W}

Candidate keys are {WX, XY}

Algorithm:

* For the first FD WX -> Y, since WX is a key for R, it does not violate BCNF.
* Second FD, X -> Z violates BCNF as X is not a key. Thus we create two relations (XZ) and (WXY).

Now inspect the given two relations for BCNF,

For (XZ), the candidate key is X. The only FD that applies here is X -> Z, so it is in BCNF.

For (WXY), the candidate keys are WX and XY.

* The first FD applies, WX -> Y, and WX is a key, so it is in BCNF.
* The second FD doesn’t apply (there is any Z in it).
* The third FD, Y->W, Y, is not a super key; thus, we need to decompose by creating a new relation.

(XZ)(YW)(XY)

* XZ: The XZ is still in BCNF as before.
* YW: The YW has Y as the candidate key, and the only FD that applies is Y-> W. It is in BCNF.
* XY: The XY has XY as the candidate key, and no FDs apply, so it is in BCNF.

Thus our final decomposition is:

(XZ)(YW)(XY)

**Advantages of BCNF:**

1. It is a more restricted form of normalization so that the database does not end in anomalies.
2. The business rules expressed in functional dependencies are enforced using keys, and BCNF ensures that it is correctly followed.

**Fourth Normal Form(4NF):**

The fourth normal form is the next level of normalization after BCNF which deals with a more universal type of dependency known as multi-valued dependency. The fourth normal form just like the other normal forms brings in features that organize the data and eliminates anomalies and redundancy. Each normalization has a set of rules that should be followed in creating the database. And in order toBased on the type of normalization that is being wished to the larger tables are divided into smaller ones. That is referred to as the decomposition of tables. A relation should satisfy the below two conditions to be in 4NF. The conditions are:

* It should be in BCNF.
* There should be no Multi-valued Dependency.

**How does Fourth Normal Form Work?**

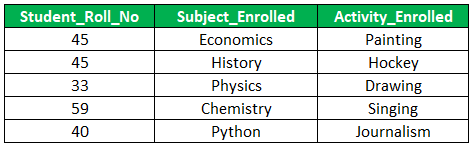
To understand how 4NF works it is necessary to understand multi-valued dependency. Multivalued dependency requires a minimum of three columns in which there should be at least two attributes that depend on the third one. And those two attributes are dependent on each other. Conditions for Multi-valued dependency.

* There should be at least 3 columns in a table.
* For every dependency A-> B, for every value of A multiple values of B exists then the dependency is referred to as a multi-valued dependency.
* In the relation of 3 columns R(XYZ), if there exists a multi-valued dependency between X and Y then Y and Z should be independent of each other.

All the above conditions should be satisfied to establish the fact that multi-valued dependency exists in relation.

**Example #1**

Here we have a Students table which has information about the subjects and the activity the students have enrolled in. Subjects\_Enrolled and Activty\_Enrolled are the two attributes that are independent of each other. The given table is in BCNF which satisfies the first condition of 4NF.



Let’s check further the Multi-valued Dependency.

1. The dependencies in this relation are:

Student\_Roll\_No —>Subject\_Enrolled

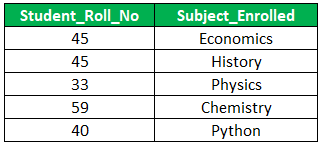
Student\_Roll\_No —> Activity Enrolled

1. Based on the conditions for Multi-Valued dependency, checking the existing relation.
2. There should be at least 3 columns in a table. – – Satisfied
3. For every dependency A-> B, for every value of A multiple values of B exists then the dependency is referred to as multi-valued dependency. – – Roll no 45 has enrolled in Economics and History in terms of academics and Painting and Hockey as activities. Thus for a value of Student\_Roll\_No different values of Activity\_Enrolled exist.
4. In the relation of 3 columns R(XYZ), if there exists a multi-valued dependency between X and Y then Y and Z should be independent of each other. – – Subject\_Enrolled and Activity Enrolled are independent of each other.

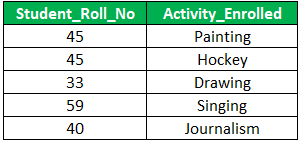
As we checked the above conditions it is clear that the relation consists of multi-valued dependency. In order to normalize the table into 4NF, we need to decompose it into smaller tables.

Student relation is now been decomposed into smaller tables S1 and S2 which are in 4NF

S1:



S2:



**Advantages of Fourth Normal Form**

Following are the advantages given.

* Helps in removing redundancy and anomalies in the database.
* Data integrity and consistency can be maintained through normalization and restricted constraints.

**Fifth normal form (5NF):**

Fifth normal form (5NF) is also known as project-join normal form (PJ/NF). It is designed to minimize redundancy in relational databases by separating semantically connected relationships in multiple formats to store multi-valued facts.

A relation R is in 5NF if and only if every non-trivial join dependency in R is implied by the candidate keys of R. A relation break up into two relations must contain lossless join Property, which makes certain that no invalid or extra tuples of attributes are created when relations are again joined together through a natural join.

**Properties:**

A relation R with attributes, its values and tuples is in 5NF if and only if the following conditions are satisfied,

* The relation R should be already in 4NF.
* The relation R cannot be additionally non loss decomposed (join dependency).

If the relation or table can further decompose to remove redundancy and anomaly, and when the process of rejoining followed the decomposed tables through the means of candidate keys, we should not be losing the original data or any new record set should not arise. Understandably, joining two or more decomposed table should not lose records or create new records.

**Join dependency:**

A table or relation can be recreated or re-designed by joining multiple tables and each table of this contain a subset of the attributes and values of the table, then the table is in Join Dependency. It is a generalization of Multivalued Dependency.

If the join of R1 and R2 over S is equal to relation R then we can say that a join dependency exists, where R1 and R2 are the decomposition R1 (P, Q, S) and R2 (Q, T) of a given relation R (P, Q, S, T). As a possibility, R1 and R2 are a lossless decomposition of R.

Over a relation R a Join Dependency ⋈ {R1, R2, ..., Rn} is said to hold if R1, R2, ..., Rn is lossless-join decomposition. The \*(P, Q, S, T), (S, T) will be a Join Dependency of R if the join of join's attribute is equal to the relation R. Here, \*(R1, R2, R3) is used to specify that relation R1, R2, R3 and so on is a Join Dependency of R.

Join Dependency can be associated to 5NF, wherein a relation is in 5NF, only if it is already in 4NF and it cannot be decomposed further.

Contrarily, in the case of functional dependencies, there is no sound and complete axiomatization for join dependencies, however, axiomatization exists for more expressive dependency languages such as full typed dependencies.

On the other hand, the implication of join dependencies is decidable.

**Example:**

The below relation violates the Fifth Normal Form (5NF) of Normalization −

**<Employee>**

|  |  |  |
| --- | --- | --- |
| **EmpName** | **EmpSkills** | **EmpJob**(Assigned Work) |
| David | Java | E145 |
| John | JavaScript | E146 |
| Jamie | jQuery | E146 |
| Emma | Java | E147 |

The above relation can be decomposed into the following three tables; therefore, it is not in 5NF −  
  
**<EmployeeSkills>**

|  |  |
| --- | --- |
| **EmpName** | **EmpSkills** |
| David | Java |
| John | JavaScript |
| Jamie | jQuery |
| Emma | Java |

The following is the <EmployeeJob> relation that displays the jobs assigned to each employee −

**<EmployeeJob>**

|  |  |
| --- | --- |
| **EmpName** | **EmpJob** |
| David | E145 |
| John | E146 |
| Jamie | E146 |
| Emma | E147 |

Here is the skills that are related to the assigned jobs −

**<JobSkills>**

|  |  |
| --- | --- |
| **EmpSkills** | **EmpJob** |
| Java | E145 |
| JavaScript | E146 |
| jQuery | E146 |
| Java | E147 |

Our Join Dependency −

|  |
| --- |
| **{(EmpName, EmpSkills ), (EmpName, EmpJob), (EmpSkills, EmpJob)}** |

The above relations have join dependency, so they are not in 5NF. That would mean that a join relation of the above three relations is equal to our original relation **<Employee>**.