DBMS 2: Functional Dependencies + Anomalies + Normalization

Functional Dependency

Functional Dependency is a set of constraints between two attributes in a relation. Functional Dependency is represented by an arrow sign (→) i.e. X → Y, where X functionally determines Y. If the value of X is known we can uniquely identify the value of Y.
Here, X (left side) is **determinant** and Y (right side) is **dependent**.

Eg:
1. **Contact_Number → User_Name**, if we know the contact number of a user then we can uniquely identify the user name.
2. **User_Id → User_Name, Contact_Number, Email_Id** - if we know the user id of a user then we can uniquely identify the user name, contact number & email id.
3. **Pincode → City,** etc.

Rules for Functional Dependencies:

1. **Reflexive Rule:** If **X1** is a subset of **X** then **X → X1.**
   Eg: X = {User_Name, Contact_Number, User_Email}
   X → Contact_Number (X1)
   X → User_Name, Contact_Number (X1) ... and so on.

2. **Augmentation Rule:** If **X** → **Y** then **XZ** → **YZ** for any **Z.**
   Eg: X = {User_Id},   Y = {User_Name},   Z = {Contact_Number}
   If User_Id → User_Name then
   User_Id, Contact_Number → User_Name, Contact_Number

3. **Transitive Rule:** If **X** → **Y** & **Y** → **Z** then **X** → **Z.**
   Eg: X = {User_Id},   Y = {Contact_Number},   Z = {User_Name}
   If User_Id → Contact_Number & Contact_Number → User_Name then
   User_Id → User_Name.

4. **Union Rule:** If **X** → **Y** and **X** → **Z** then **X** → **Y, Z.**
   Eg: X = {User_Id},   Y = {Contact_Number},   Z = {User_Name}
   If User_Id → Contact_Number & User_Id → User_Name then

User_Id → Contact_Name, User_Name.

5. **Decomposition Rule / Project Rule:** (Reverse of Union Rule).
   If **X → Y, Z** then **X → Y** and **X → Z.**
   Eg: X = {User_Id},   Y = {Contact_Number},   Z = {User_Name}
   If User_Id → Contact_Number, User_Name then
   User_Id → Contact_Number & User_Id → User_Name.

6. **Pseudo Transitive Rule:** If **X → Y** and **YZ → W** then **XZ → W.**
   Eg: X = {User_Id}, Y = {Contact_Number}, Z = {User_Name}, W = {User_Email}
   If User_Id → Contact_Number and
   Contact_Number, User_Name → User_Email then
   User_Id, User_Name → User_Email.

<u>Types of Functional Dependencies:</u>

Trivial: If in X → Y, Y is a subset of X then it's called a trivial functional dependency. Eg: User_Id, User_Name → User_Name.	**Non-Trivial:** If in X → Y, Y is not a subset of X then it's called a non-trivial functional dependency. Eg: User_Id, User_Name → User_Name, User_Email	**Completely Non-Trivial:** If in X → Y, X intersection Y = {} (empty) then it's called completely non-trivial functional dependency. Eg: User_Id → User_Email, User_Name

If a database design is not perfect, it may contain anomalies (problems), which are like a bad dream for any database administrator. Managing a database with anomalies is next to impossible.

<u>The Problem of redundancy in Database</u>

Redundancy means having **multiple copies of the same data** in the database. This problem arises when a database is not normalized. Suppose in a college, a table of student details attributes are *student Id, student name, contact number, college name, course opted & **college rank**.*

Student_ID	Name	Contact	College	Course	Rank
100	Himanshu	7300934851	GEU	Btech	1
101	Ankit	7900734858	GEU	Btech	1
102	Aysuh	7300936759	GEU	Btech	1
103	Ravi	7300901556	GEU	Btech	1

As it can be observed that values of attribute college name, college rank, the course is being repeated which can lead to problems. **Problems caused due to redundancy are Insertion anomaly, Deletion anomaly, and Updation anomaly.**

1. **Insertion Anomaly –**
   a. If a student detail has to be inserted whose course is not being decided yet then insertion of complete data will not be possible till the time course is decided for the student.
   b. Although we have an option of inserting incomplete data, that can cause problems if the data is used before adding complete data.
   c. **For eg: If we query to count the number of students who are in a particular course,** this student will not be counted in any course but ideally, the sum of students in all courses should be equal to the total number of students in the college.
   d. This problem happens when the insertion of a data record is not possible without adding some additional unrelated data to the record, eg: adding NULL in the course column.

Student_ID	Name	Contact	College	Course	Rank
100	Himanshu	7300934851	GEU		1

2. **Deletion Anomaly –**
   a. If the current batch graduates and we delete all the data of the students then the details of college (GEU college has rank 1) will also get deleted which should not occur ideally.

b. This anomaly happens when deletion of a data record results in losing some unrelated information that was stored as part of the record that was deleted from a table.

3. **Update Anomaly –**
   a. Suppose if the rank of the college changes then changes will have to be all over the database which will be time-consuming and computationally costly.
   b. If the update does not occur at all places then the database will be in an inconsistent state.

Student_ID	Name	Contact	College	Course	Rank
100	Himanshu	7300934851	GEU	Btech	1
101	Ankit	7900734858	GEU	Btech	1
102	Aysuh	7300936759	GEU	Btech	1
103	Ravi	7300901556	GEU	Btech	1

All places should be updated

<u>Normalization</u>

- Normalization is the process of organizing the data in the database.
- Normalization is used to minimize the redundancy from a relation or set of relations. It is **a method to remove all the anomalies and bring the database to a consistent state**.

- Normalization divides the larger table into the smaller table and links them using relationships.
- Normalization is a stepwise process and it goes from First Normal Form → Second Normal Form → Third Normal form → BC Normal Form.

First Normal form (1NF):

Each attribute should have an atomic value, **multi-value attributes are not allowed**.

If one user has **multiple contact numbers** then storing it in one column looks like

User_Id	User_Name	User_Contact
1234	Utkarsh	9876598765, 9898765765
2468	Karan	9876543210, 9898989898, 9897969594

To convert this data in **1NF** we have two options:

1. Have **multiple columns** for contact numbers like:

   User_Contact_1, User_Contact_2, and User_Contact_3.

User_Id	User_Name	User_Contact_1	User_Contact_2	User_Contact_3
1234	Utkarsh	9876598765	9898765765	NULL
2468	Karan	9876543210	9898989898	9897969594

Que: What are the problems that can happen due to this?

   But this will:

   a. If there are multiple users who don't have 3 contact numbers then most of the cells will be marked NULL, wasting some space that could be avoided.
   b. Restricting the user who has more than 3 contact numbers as adding a new column for only a single user is not ideal because that column will have NULL for all other users.

2. Have **multiple rows** to store different contact numbers.

User_Id	User_Name	User_Contact
1234	Utkarsh	9876598765
1234	Utkarsh	9898765765
2468	Karan	9876543210
2468	Karan	9898989898
2468	Karan	9897969594

Que: What are the problems that can happen due to this?

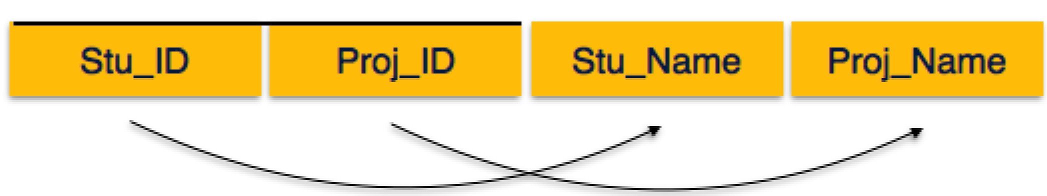
But this will increase the redundancy & waste some space as we are storing User_Id and User_Name multiple times - But this can be taken care of as we go to **2NF**.

Second Normal Form (2NF):

Every **non-prime** attribute should be fully functionally dependent on the **prime** key attribute. That is, if **X → A** holds, then there should not be any **proper subset Y of X**, for which **Y → A** also holds true.

Prime key Attribute: Attributes that are part of one of the candidate keys.

Student_Project



W

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

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

Student

Stu_ID	Stu_Name	Proj_ID

Project

Proj_ID	Proj_Name

Que: What is the need for Proj_ID in the Student table?

Ans: Proj_ID is a **foreign key** in the Student table that is creating a link between the two tables.

Student_Id	**Project_Id**	**Student_Name**	**Project_Name**

If we have **Many :: Many → Project :: Student** then having **Student_Id** in ***Project Table*** or having **Project_ID** in ***Student table*** will make it a **multi-valued attribute**, **hence we need 3 tables where the third table will be a relationship table.**

Student Table – **[Student_Id, Student_Name]**

Project Table – **[Project_Id, Project_Name]**

Student_Project Table – **[Student_Id, Project_Id]**

Third Normal Form (3NF):

No **non-prime** attribute is **transitively dependent** on the **prime** key attribute.

For every non-trivial functional dependency, **X → Y**, either **X** is a super key or **Y** is a prime attribute.

In simple words, if we have a transitive dependency in the **User** table, where **User_Id** is the primary key, like:

User_Id	User_Zip	User_City

Here, User_Id → User_Zip & User_Zip → User_City therefore, by Transitive Dependency **User_Id → User_Zip → User_City.**

In this case, **neither User_Zip is a super key** (as it cannot identify any row uniquely - there can be multiple users living in the same zip location) nor **User_City** is a prime attribute (not part of any candidate key). Hence, there is a transitive partial dependency that violates **3NF.**

To convert this into **3NF** keep all the user data in the **User** table except **User_City,** and have a separate table of **[Zip, City]** where Zip is a primary key.

User Table

User_Id	User_Zip

City Table

Zip	City

<u>Boyce-Codd Normal form (BCNF)</u>:

It is an **extension of 3NF** which states that for any **non-trivial** functional dependency, **X → Y**, **X** must be a super key.

For eg: In the above example, **Zip → City** where **Zip** was a super key, and **User_Id → Zip** where **User_Id** was a super key, therefore it was in **BCNF** form as well.

Consider this table:

<u>**User_Id**</u>	**User_Name**	**Country_Code**	**Country_Name**
1234	Utkarsh	+91	India
5678	Abhishek	+91	India
2468	Karan	+1	US

In this table we have a functional dependency **Country_Code → Country_Name** in which **Country_Code** is not a super key (as it cannot find a row uniquely) hence this table is not in **BCNF.**

To convert the above data into BCNF**,** have separate tables for User data that does not have Country_Name, and have a separate table for Country data.

<u>**User_Id**</u>	**User_Name**	**Country_Code**
1234	Utkarsh	+91
5678	Abhishek	+91
2468	Karan	+1

<u>**Country_Code**</u>	**Country_Name**
+91	India → Bharat
+1	US

<u>Summary:</u>

1NF = No multi-valued attribute allowed.

In every non-trivial functional dependency, **X → Y:**

2NF = No partial dependency

3NF = No transitive dependency

BCNF = **X** must be a super key.

Denormalization

From a purist point of view, you want to normalize your data as much as possible, but from a practical point of view you will find that you need to 'back out' of some of your normalizations for performance reasons. This is called "denormalization".