UNIT III

DATABASE DESIGN & APPLICATION DEVELOPMENT

In the context of **Database Management Systems (DBMS)**, **dependencies** refer to relationships between different attributes or columns in a database table. These dependencies are used to define how the values in one or more attributes determine the values in another attribute. Understanding dependencies helps in database normalization, ensuring the structure is efficient, free from redundancy, and can be easily maintained.

Types of Dependencies in DBMS

1. **Functional Dependency (FD)**:
   - A functional dependency occurs when one attribute or a set of attributes in a table uniquely determines the value of another attribute.
   - **Notation**: If **A → B**, it means that attribute **A** functionally determines attribute **B**.
   - **Example**:
     - Consider a table `Employee (EmpID, EmpName, EmpDept)`.
     - If **EmpID → EmpName**, it means that for each **EmpID**, there is a unique **EmpName** associated with it.
     - **EmpID** determines **EmpName** because a specific employee ID corresponds to only one employee name.
2. **Partial Dependency**:
   - A partial dependency occurs when a non-prime attribute is dependent on part of a composite primary key (i.e., a key that consists of more than one attribute).
   - **Example**:
     - Consider a table `Course (CourseID, StudentID, InstructorName, InstructorPhone)`, where the primary key is `(CourseID, StudentID)`.
     - If **CourseID → InstructorName**, then **InstructorName** depends only on **CourseID**, which is a part of the composite primary key. This is a partial dependency because it doesn't depend on the whole key.
3. **Transitive Dependency**:
   - A transitive dependency occurs when one attribute depends on another attribute, which in turn depends on a third attribute. In other words, if **A → B** and **B → C**, then **A → C** is a transitive dependency.
   - **Example**:
     - Consider a table `Student (StudentID, StudentName, CourseName, InstructorName)`.
     - If **StudentID → StudentName** and **StudentName → InstructorName**, then **StudentID → InstructorName** is a transitive dependency.
4. **Multivalued Dependency (MVD)**:
   - A multivalued dependency occurs when one attribute in a table determines multiple values for another attribute, and these multiple values are independent of each other.
   - **Example**:
     - Consider a table `Employee (EmpID, Skill, Language)`.
     - If an employee can have multiple skills and languages, then **EmpID →→ Skill** and **EmpID →→ Language** represent multivalued dependencies, meaning the skills and languages are independent but both depend on the employee ID.

5. **Join Dependency**:
    - A join dependency occurs when a table can be split into two or more smaller tables, and the original table can be recreated by performing a join operation on those smaller tables.
    - **Example**:
        - If a table `EmployeeProject (EmpID, ProjectID)` can be split into two smaller tables like `Employee(EmpID)` and `Project(ProjectID)`, then a join dependency exists between these tables.
6. **Boyce-Codd Normal Form (BCNF)**:
    - A special case of functional dependency. A table is in BCNF if for every functional dependency **A → B**, **A** is a superkey.
    - **Example**:
        - A table `Student (StudentID, CourseID, InstructorID)` with dependencies:
            - **StudentID → CourseID**
            - **InstructorID → CourseID**
        - The second dependency **InstructorID → CourseID** violates BCNF because **InstructorID** is not a superkey.

Conclusion

In DBMS, dependencies help define how attributes relate to each other, guiding the normalization process to minimize redundancy and ensure data integrity. By understanding different types of dependencies (like functional, partial, transitive, and multivalued), we can design better, more efficient database schemas.

A deletion anomaly occurs in a database when deleting one piece of information causes the unintended loss of other related, necessary data. This usually happens in poorly normalized databases where related data is stored together in a single table, and deleting one part can lead to the loss of other essential information.

Example of Deletion Anomaly:

Consider a database table that stores information about students and the courses they are enrolled in:

Student_ID	Student_Name	Course_Name	Professor_Name
101	Alice	Math	Dr. Smith
102	Bob	Physics	Dr. Jones
101	Alice	Physics	Dr. Jones
103	Charlie	Math	Dr. Smith

In this table, the information about each student's enrollment is stored, including the student name, course name, and professor's name.

Now, imagine if **Alice** decides to drop the **Math** course. To remove her enrollment from the **Math** course, we would delete the following row:

| 101 | Alice | Math | Dr. Smith |

However, if **Dr. Smith** is the professor for only **Alice's Math** course, deleting this row removes all information about **Dr. Smith**, including the fact that he is the professor for **Alice's** other course, **Physics**. This results in a **deletion anomaly**, where removing a single row (Alice's Math enrollment) also removes important information about the professor (Dr. Smith) that might be necessary for other students enrolled in the same course (e.g., **Charlie**).

Consequence:

- The professor's information is lost from the database, even though Dr. Smith is still the professor for other courses (e.g., Alice's Physics course).
- This can cause inconsistency in the database because now we don't have a clear record of who the professor is for the **Physics** course anymore.

Solution:

To avoid this anomaly, the database should be **normalized**. This can be done by splitting the data into separate tables to reduce redundancy and avoid losing necessary information when deleting records:

1. **Student Table:**
    - Student_ID, Student_Name
2. **Course Table:**
    - Course_Name, Professor_Name
3. **Enrollment Table:**
    - Student_ID, Course_Name

By normalizing the database, deleting a student's enrollment from a course will not result in the loss of professor information, as the professor data is stored in a separate table.

An **updation anomaly** occurs when the same piece of data is stored in multiple places (redundancy), and when an update is made to one instance of that data, it is not reflected in other instances, leading to inconsistency. This happens when data is repeated in different places but is not properly related, typically in a poorly normalized database.

Example of Updation Anomaly:

Consider a database table that stores information about employees and the departments they belong to:

Employee_ID	Employee_Name	Department_Name	Department_Location
101	John	HR	New York
102	Sarah	Finance	London
103	Mark	HR	New York
104	Anna	Marketing	Paris
105	Paul	Finance	London

In this table:

- The **Department_Name** and **Department_Location** are repeated for multiple employees.
- For example, **John** and **Mark** both belong to the **HR** department, which is located in **New York**.
- **Sarah** and **Paul** belong to the **Finance** department, which is located in **London**.

Scenario: Updation Anomaly

Let's say that the **HR** department moves its office from **New York** to **San Francisco**. Now, we need to update the **Department_Location** for all employees in the HR department.

If we only update the row for **John** (e.g., change the location to "San Francisco") and forget to update the row for **Mark**, the data becomes inconsistent:

Employee_ID	Employee_Name	Department_Name	Department_Location
101	John	HR	San Francisco
102	Sarah	Finance	London
103	Mark	HR	New York
104	Anna	Marketing	Paris
105	Paul	Finance	London

In this case:

- The **HR department** is listed as being in **San Francisco** for **John** but still listed as being in **New York** for **Mark**, even though both belong to the same department.
- This results in an **updation anomaly**, where the same data (department location) is not updated consistently across all records, leading to **data inconsistency**.

Solution:

To avoid an updation anomaly, the database should be normalized. The solution is to separate the employee data from department data, which eliminates redundancy and ensures that updates can be made in one place.

1. **Employee Table:**
    - Employee_ID, Employee_Name, Department_ID
2. **Department Table:**

- Department_ID, Department_Name, Department_Location

By normalizing the database, you would store the department information in one table, and refer to it in the employee table through a **Department_ID**. Updating the location of the **HR** department in the **Department Table** would automatically reflect for all employees in that department without having to update each employee's record.

Example of normalized tables:

Employee Table:

Employee_ID	Employee_Name	Department_ID
101	John	1
102	Sarah	2
103	Mark	1
104	Anna	3
105	Paul	2

Department Table:

Department_ID	Department_Name	Department_Location
1	HR	New York
2	Finance	London
3	Marketing	Paris

Now, if the **HR department** moves to **San Francisco**, you only need to update the **Department Table** once, and all employees in the **HR department** (John and Mark) will reflect the updated location automatically, avoiding the updation anomaly.

An **insertion anomaly** occurs when we are unable to insert data into a database due to the absence of certain required attributes or data that may not be available at the time of insertion. This typically happens in a poorly normalized database where multiple pieces of information are stored together, and inserting data requires filling in unrelated or irrelevant fields, which can be problematic when some data is missing.

Example of Insertion Anomaly:

Consider a database table that stores information about **employees** and the **projects** they are assigned to:

Employee_ID	Employee_Name	Project_Name	Project_Deadline
101	John	Website Redesign	2024-12-15
102	Sarah	Mobile App	2024-11-30

In this table, every time we insert an employee's project details, we need to fill in both the **employee's information** (name, ID) and the **project details** (name, deadline).

Scenario: Insertion Anomaly

Now, suppose we want to insert a new **employee**, **Mark**, who has not yet been assigned to a **project**. We might try to insert his details like this:

Employee_ID	Employee_Name	Project_Name	Project_Deadline
103	Mark	NULL	NULL

However, because the **Project_Name** and **Project_Deadline** fields are required (i.e., they cannot be left empty or `NULL` due to database constraints), we are **unable to insert Mark's record** into the table, even though the employee exists and should be part of the database.

Why This Happens:

- The table design forces us to insert **project-related data** even though **Mark** may not yet be assigned to any project.
- This creates an **insertion anomaly** because **Mark**'s data cannot be inserted due to the absence of project details, even though we don't yet have any relevant project information for him.

Solution:

To solve this problem, we can **normalize** the database by separating the **employee data** and **project data** into different tables, and linking them with a relationship table. This way, we can insert employee data without needing project details right away, and then associate the employee with a project when the information becomes available.

1. **Employee Table:**
    - Employee_ID, Employee_Name
2. **Project Table:**
    - Project_ID, Project_Name, Project_Deadline
3. **Employee_Project Table (Relationship Table):**
    - Employee_ID, Project_ID

Now, we can insert **Mark's data** into the **Employee Table** without worrying about the **project details**:

Employee Table:

Employee_ID	Employee_Name
101	John
102	Sarah
103	Mark

When Mark is assigned to a project, we can insert the relevant data into the **Employee_Project Table**:

Employee_Project Table:

Employee_ID	Project_ID
103	2

This ensures that we can insert employee records even if they are not yet assigned to a project, avoiding the **insertion anomaly**.

1NF (First Normal Form) is a property of a relational database table that ensures the following:

1. **Atomicity**: Each column contains atomic (indivisible) values, meaning no multiple values or sets of values are stored in a single column.
2. **Uniqueness**: Each record (row) in the table must be unique, and there should be no duplicate rows.
3. **No repeating groups**: Each column must contain unique, single values, and there should be no repeating groups or arrays within a column.

Example of a Table Not in 1NF:

Consider the following table that stores information about students and the courses they are enrolled in:

Student_ID	Student_Name	Courses
101	Alice	Math, Physics
102	Bob	Chemistry, Biology
103	Charlie	Math, Chemistry

In this table:

- The **Courses** column contains **multiple values** for each student (e.g., Alice has "Math, Physics" in one column, and Bob has "Chemistry, Biology").
- This violates **1NF**, because each column must contain atomic values, and the **Courses** column should not contain multiple values.

Converting the Table to 1NF:

To convert the table to **1NF**, we must ensure that each column contains **only atomic values**. So, we will create a new row for each course a student is enrolled in.

The table in **1NF** will look like this:

Student_ID	Student_Name	Course
101	Alice	Math

Student_ID	Student_Name	Course
101	Alice	Physics
102	Bob	Chemistry
102	Bob	Biology
103	Charlie	Math
103	Charlie	Chemistry

Why This is Now in 1NF:

- The **Courses** column has been split into individual rows, ensuring each column holds only a single value per row.
- There are no multiple values in any column (atomicity is preserved).
- The table now satisfies **1NF**, as each column contains atomic values and there are no repeating groups or sets of values.

Summary:

The original table was **not in 1NF** because the **Courses** column contained multiple values for each student. After splitting the data, the table now follows **1NF**, with each row representing a single atomic value for each student-course relationship.

2NF (Second Normal Form) builds upon the rules of **1NF (First Normal Form)** and addresses **partial dependencies**. A table is in **2NF** if:

1. It is in **1NF**.
2. It does not have **partial dependencies**. That means every non-prime attribute (non-key attribute) must be fully dependent on the entire primary key, not just part of it.

A **partial dependency** occurs when a non-key attribute depends only on part of a composite primary key rather than the whole key.

Example of a Table Not in 2NF:

Let's consider a table that stores information about student enrollments and courses:

Student_ID	Course_ID	Student_Name	Course_Name	Instructor
101	301	Alice	Math	Dr. Smith
101	302	Alice	Physics	Dr. Jones
102	301	Bob	Math	Dr. Smith
103	303	Charlie	Chemistry	Dr. White

In this table:

- **Primary Key**: The combination of **Student_ID** and **Course_ID** uniquely identifies each row because a student can take multiple courses, and each course can have multiple students.
- The non-key attributes are **Student_Name**, **Course_Name**, and **Instructor**.

Why This Table Is Not in 2NF:

- The table is in **1NF**, as it contains atomic values.
- However, there are **partial dependencies**:
  - **Student_Name** depends only on **Student_ID**, not the whole composite key (**Student_ID, Course_ID**).
  - **Course_Name** and **Instructor** depend only on **Course_ID**, not the whole composite key (**Student_ID, Course_ID**).

Since these non-key attributes depend only on part of the primary key, the table violates **2NF**.

Converting the Table to 2NF:

To bring the table into **2NF**, we need to remove partial dependencies. We can do this by creating separate tables for **students**, **courses**, and **enrollments**, where each non-key attribute is fully dependent on the primary key.

1. **Student Table** (Stores student-specific information):

Student_ID	Student_Name
101	Alice
102	Bob
103	Charlie

2. **Course Table** (Stores course-specific information):

Course_ID	Course_Name	Instructor
301	Math	Dr. Smith
302	Physics	Dr. Jones
303	Chemistry	Dr. White

3. **Enrollment Table** (Stores which student is enrolled in which course):

Student_ID	Course_ID
101	301
101	302
102	301
103	303

Why This Table Is Now in 2NF:

- Each table is in **1NF**, as all values are atomic.
- In the **Student Table**, **Student_Name** depends entirely on **Student_ID** (a single attribute primary key).
- In the **Course Table**, both **Course_Name** and **Instructor** depend entirely on **Course_ID** (a single attribute primary key).
- In the **Enrollment Table**, the **Student_ID** and **Course_ID** together act as the composite primary key, and there are no partial dependencies.

Now, there are no non-key attributes that depend on only part of the primary key. Each non-key attribute is fully functionally dependent on the entire primary key, satisfying **2NF**.

Summary:

The original table was **not in 2NF** because there were partial dependencies (e.g., **Student_Name** depends only on **Student_ID**, and **Course_Name** depends only on **Course_ID**). After splitting the table into **three tables**, each non-key attribute is fully dependent on the whole primary key, and the database is now in **2NF**.

3NF (Third Normal Form) builds upon the rules of **2NF (Second Normal Form)** and addresses **transitive dependencies**. A table is in **3NF** if:

1. It is in **2NF**.
2. It has no **transitive dependencies**. This means that non-prime attributes (attributes that are not part of the primary key) should not depend on other non-prime attributes. Every non-prime attribute should depend only on the **primary key** and not on other non-prime attributes.

Example of a Table Not in 3NF:

Let's consider a table storing information about **students**, the **courses** they are enrolled in, and their respective **instructors**:

Student_ID	Course_ID	Student_Name	Course_Name	Instructor	Instructor_Phone
101	301	Alice	Math	Dr. Smith	123-456-7890
101	302	Alice	Physics	Dr. Jones	987-654-3210
102	301	Bob	Math	Dr. Smith	123-456-7890
103	303	Charlie	Chemistry	Dr. White	555-123-4567

Why This Table is Not in 3NF:

- The table is in **2NF** because it satisfies both **1NF** and **2NF** (no partial dependencies).
- However, there is a **transitive dependency**:
  - **Instructor_Phone** depends on **Instructor**, which is not a primary key but a non-prime attribute.

- **Instructor** itself depends on **Course_ID** (since each course has a specific instructor), and therefore **Instructor_Phone** is transitively dependent on **Course_ID** through **Instructor**.

This creates redundancy because the **Instructor_Phone** number is repeated for every row where the same **Instructor** is teaching the course.

Converting the Table to 3NF:

To convert the table to **3NF**, we need to remove the transitive dependency. This can be done by separating the **Instructor** and **Instructor_Phone** information into a new table, and linking it with the **Course** table.

1. **Student Table** (stores student information):

Student_ID	Student_Name
101	Alice
102	Bob
103	Charlie

2. **Course Table** (stores course information, including the instructor):

Course_ID	Course_Name	Instructor_ID
301	Math	1
302	Physics	2
303	Chemistry	3

3. **Instructor Table** (stores instructor information):

Instructor_ID	Instructor	Instructor_Phone
1	Dr. Smith	123-456-7890
2	Dr. Jones	987-654-3210
3	Dr. White	555-123-4567

4. **Enrollment Table** (stores which students are enrolled in which courses):

Student_ID	Course_ID
101	301
101	302
102	301
103	303

Why This Table is Now in 3NF:

- The database is now in **2NF**, as we eliminated partial dependencies by separating the student, course, and instructor information into distinct tables.
- The table is in **3NF** because there are no **transitive dependencies**:
    - **Instructor_Phone** now depends directly on the **Instructor_ID** in the **Instructor Table**, which is the primary key for that table.
    - **Instructor** depends on **Instructor_ID**, and **Instructor_Phone** depends on **Instructor_ID**, so no non-prime attribute depends on another non-prime attribute in any table.

In essence, we've removed the indirect relationship between **Instructor_Phone** and **Course_ID** that existed through **Instructor**.

Summary:

The original table was **not in 3NF** because it had a **transitive dependency** (Instructor_Phone → Instructor → Course_ID). After normalizing the database into four separate tables, the transitive dependency was eliminated, ensuring that all non-key attributes depend directly on the primary key, and the database is now in **3NF**.

BCNF (Boyce-Codd Normal Form) is a higher level of normalization that builds upon **3NF (Third Normal Form)**. A table is in **BCNF** if:

1. It is in **3NF**.
2. For every **non-trivial functional dependency** X→YX \to YX→Y, XXX must be a **superkey**. A **superkey** is a set of attributes that uniquely identifies a record in a table. In other words, if a non-prime attribute depends on another attribute, that attribute must be a superkey.

This means **BCNF** removes any remaining functional dependencies where non-prime attributes are dependent on non-superkey attributes, which might still be allowed in **3NF**.

Example of a Table Not in BCNF:

Let's consider a table that stores information about **students**, **courses**, and **instructors**:

Student_ID	Course_ID	Instructor_ID	Instructor_Name
101	301	1	Dr. Smith
102	301	1	Dr. Smith
103	302	2	Dr. Jones
104	303	3	Dr. White

Functional Dependencies:

- **Student_ID, Course_ID → Instructor_ID, Instructor_Name** (A combination of Student_ID and Course_ID uniquely determines the instructor's details).
- **Instructor_ID → Instructor_Name** (Each instructor has a unique name).

Why This Table is Not in BCNF:

- The table is in **3NF** because there are no transitive dependencies, and all non-key attributes depend on the entire primary key.
- However, there is a **functional dependency** where **Instructor_ID → Instructor_Name**, but **Instructor_ID** is **not a superkey** (since **Student_ID, Course_ID** is the primary key). This violates BCNF, as **Instructor_ID** is not a superkey, but it determines the **Instructor_Name**.

Converting the Table to BCNF:

To make the table **BCNF compliant**, we need to separate the instructor information into its own table, ensuring that **Instructor_ID** is a superkey in its own table.

We can split the data into two tables:

1. **Instructor Table** (stores instructor details):

Instructor_ID	Instructor_Name
1	Dr. Smith
2	Dr. Jones
3	Dr. White

2. **Enrollment Table** (stores student-course relationships):

Student_ID	Course_ID	Instructor_ID
101	301	1
102	301	1
103	302	2
104	303	3

Why This is Now in BCNF:

- The **Instructor Table** satisfies **BCNF** because **Instructor_ID** is now a superkey in that table.
- The **Enrollment Table** still has **Student_ID, Course_ID** as the primary key, and there is no violation of BCNF because no non-key attribute depends on anything other than the superkey.

By separating the data into two tables, we eliminated the non-superkey dependency (where **Instructor_ID → Instructor_Name**), ensuring the database is now in **BCNF**.

Summary:

The original table violated **BCNF** because **Instructor_ID** determined **Instructor_Name**, but **Instructor_ID** was not a superkey. After splitting the table into two tables—one for **Instructor** and one for **Enrollment**—we ensured that every non-trivial functional dependency is based on a superkey, thus bringing the database into **BCNF**.

Multi-Valued Dependencies (MVD) and Fourth Normal Form (4NF)

4NF (Fourth Normal Form) is a higher level of normalization that deals with **multi-valued dependencies (MVDs)**. A table is in **4NF** if:

1. It is in **BCNF** (Boyce-Codd Normal Form).
2. It has no **multi-valued dependencies**. A **multi-valued dependency** occurs when one attribute determines multiple independent values for another attribute, and those values are not related to other attributes in the table.

In other words, a table is in **4NF** if no non-trivial multi-valued dependency exists. A multi-valued dependency exists when a row in a table implies several independent facts, each of which should be represented by a separate row.

What is a Multi-Valued Dependency (MVD)?

A **multi-valued dependency (MVD)** occurs when one attribute (or set of attributes) determines multiple values for another attribute, but these multiple values are **independent** of each other. This means that there is no relationship between the multiple values for an attribute, and they should be stored separately.

Formally, if for a relation RRR, we have attributes A,B,CA, B, CA,B,C, then a multi-valued dependency A→→BA \to\to BA→→B holds if for each value of AAA, there is a set of values for BBB that is independent of the values in other attributes like CCC.

Example of a Table Not in 4NF:

Let's consider a table that stores information about **students**, their **courses**, and the **hobbies** they have:

Student_ID	Course	Hobby
101	Math	Reading
101	Physics	Painting
101	Math	Cycling

Student_ID	Course	Hobby
102	Chemistry	Gardening
102	Biology	Reading
103	History	Painting

Functional Dependencies:

- **Student_ID, Course** → (Student information and course are linked).
- **Student_ID** → (Student details are dependent on the **Student_ID**).

Why This Table is Not in 4NF:

This table contains a **multi-valued dependency**:

- **Student_ID →→ Hobby**: A student can have multiple hobbies, but the hobbies are independent of the courses the student is enrolled in.
- **Student_ID →→ Course**: A student can be enrolled in multiple courses, but the courses are independent of the hobbies.

Thus, the table violates **4NF** because we are storing both the **courses** and **hobbies** for each student in the same table, which results in redundancy. Specifically, for **Student_ID 101**, we have:

- Math → Reading, Painting, Cycling
- Physics → Reading, Painting, Cycling

Converting the Table to 4NF:

To bring this table into **4NF**, we need to eliminate the multi-valued dependencies by creating separate tables for the independent facts (courses and hobbies). Each fact (course and hobby) should be stored in a separate table with a reference to the **Student_ID**.

We split the original table into two tables:

1. **Student_Courses Table** (stores which courses a student is enrolled in):

Student_ID	Course
101	Math
101	Physics
102	Chemistry
102	Biology
103	History

2. **Student_Hobbies Table** (stores the hobbies of a student):

Student_ID	Hobby
101	Reading
101	Painting
101	Cycling
102	Gardening
102	Reading
103	Painting

Why These Tables are Now in 4NF:

- Each table is in **BCNF** because each non-key attribute is fully dependent on the primary key, and there are no non-trivial functional dependencies violating BCNF.
- Both tables are now in **4NF** because:
  - The multi-valued dependencies are eliminated. The **hobbies** of a student and the **courses** a student is enrolled in are stored independently in separate tables.
  - There is no redundancy caused by storing multiple independent values for a student in the same table.

Summary:

The original table violated **4NF** because it had multi-valued dependencies: **Student_ID →→ Hobby** and **Student_ID →→ Course**, which led to redundancy. By splitting the table into two separate tables—one for **Student_Courses** and one for **Student_Hobbies**—we eliminated the multi-valued dependencies and brought the table into **4NF**.

Join Dependencies and Fifth Normal Form (5NF)

5NF (Fifth Normal Form), also known as **Projection-Join Normal Form (PJNF)**, deals with **join dependencies**. A table is in **5NF** if:

1. It is in **4NF** (Fourth Normal Form).
2. It does not contain any **join dependencies** that are not implied by the candidate keys.

A **join dependency** occurs when a table can be reconstructed (or decomposed) into multiple tables, but the decomposition involves losing data because the relationships between attributes are not clear. In simpler terms, **5NF** requires that all data be fully reconstructed using joins without losing any information. A table is in **5NF** when it cannot be decomposed into smaller tables without losing data or introducing redundancy.

What is a Join Dependency?

A **join dependency** occurs when a table contains attributes that, when decomposed into separate tables, can still be joined back together without loss of information, but the decomposition doesn't necessarily follow from the candidate keys. A table that satisfies **5NF** doesn't have unnecessary join dependencies.

Example of a Table Not in 5NF:

Consider a table storing information about **students**, **courses**, and the **instructors** of those courses:

Student_ID	**Course_ID**	**Instructor_ID**
101	301	1
101	302	2
102	301	1
103	303	3
103	302	2

Functional Dependencies:

- **Student_ID, Course_ID → Instructor_ID** (A student is enrolled in a specific course, which is taught by a specific instructor).

Why This Table is Not in 5NF:

In this table, we have a **join dependency** because it can be decomposed into smaller tables, but this decomposition does not naturally follow from the candidate keys and may result in redundancy:

- **Student_Courses Table**:

Student_ID	**Course_ID**
101	301
101	302
102	301
103	303
103	302

- **Course_Instructors Table**:

Course_ID	**Instructor_ID**
301	1
302	2
303	3

- **Student_Course_Instructors Table**:

Student_ID	**Instructor_ID**
101	1
101	2

Student_ID	Instructor_ID
102	1
103	2
103	3

This decomposition introduces redundancy and loss of information since when the tables are joined back, we might encounter situations where the **Instructor_ID** is incorrectly paired with a **Student_ID** and **Course_ID** combination, which leads to **spurious tuples** (invalid combinations).

Converting the Table to 5NF:

To make the table comply with **5NF**, we need to break it down further to ensure there is no unnecessary decomposition and that no additional relationships are implied by the decomposition. A proper decomposition would involve separating the attributes that do not have any intrinsic relationship but are being forced into a single table, resulting in a **lossless decomposition**.

We can decompose the original table into the following smaller tables that each represent a **fact** independently:

1. **Student_Courses Table** (records which students are enrolled in which courses):

Student_ID	Course_ID
101	301
101	302
102	301
103	303
103	302

2. **Course_Instructors Table** (records which courses are taught by which instructors):

Course_ID	Instructor_ID
301	1
302	2
303	3

3. **Student_Instructors Table** (records which students are taught by which instructors):

Student_ID	Instructor_ID
101	1
101	2
102	1
103	2
103	3

Why This is Now in 5NF:

- Each of the smaller tables now represents a **single fact**, and we have eliminated any **join dependencies** that might have caused redundant data.
- These smaller tables can be **joined** back together (using **Student_ID** and **Course_ID**, or **Instructor_ID**) without creating redundant or spurious data.
- The decomposition is now lossless because each table contains only facts that are directly related, and we can always reconstruct the original table by joining the smaller tables back together.

Summary:

The original table violated **5NF** because it had a **join dependency**: it could be decomposed into smaller tables, but this decomposition did not naturally follow from the candidate keys and led to redundant data. After decomposing the table into three smaller tables (Student_Courses, Course_Instructors, and Student_Instructors), the table is now in **5NF** because the decomposition is lossless, and no further splitting is possible without losing data or creating redundancy.

Decomposition Rules and Example

Decomposition in the context of database normalization refers to breaking down a large, complex table into smaller, more manageable tables while preserving the original data and ensuring that no information is lost in the process. The goal is to achieve the desired normal form (e.g., **3NF**, **BCNF**, **4NF**, **5NF**) through decomposition.

There are certain rules that guide the decomposition of a table to achieve a higher normal form, particularly for **3NF**, **BCNF**, and **4NF**.

Decomposition Rules:

1. **Lossless Decomposition**:
    - A decomposition is **lossless** if you can recover the original relation by joining the decomposed relations.
    - This means no information is lost, and no spurious tuples are introduced when performing the join.
2. **Dependency-Preserving Decomposition**:
    - A decomposition is **dependency-preserving** if the functional dependencies that were valid in the original table still hold in the decomposed tables, so you don't lose the ability to enforce constraints.
3. **Preserving Keys**: The keys of the original table should be preserved in the decomposed tables, so that they can still be used to uniquely identify records.

Example of Decomposition:

Let's consider a table that contains information about **students**, **courses**, and **instructors**:

Student_ID	Course_ID	Instructor	Instructor_Phone	Student_Name
101	301	Dr. Smith	123-456-7890	Alice
102	301	Dr. Smith	123-456-7890	Bob
101	302	Dr. Jones	987-654-3210	Alice
103	303	Dr. White	555-123-4567	Charlie

Step 1: Identify Functional Dependencies

Based on the data, we can identify some functional dependencies:

1. **Student_ID, Course_ID → Student_Name**: A student has a specific name.
2. **Course_ID → Instructor**: A course has a specific instructor.
3. **Instructor → Instructor_Phone**: Each instructor has a specific phone number.
4. **Student_ID, Course_ID → Instructor**: A student takes a course with a specific instructor.

Step 2: Analyze Violations of Normal Forms

- **1NF**: The table is in **1NF** as it does not contain any repeating groups or nested data.
- **2NF**: The table violates **2NF** because it has partial dependencies. For example, **Instructor_Phone** is dependent on **Instructor**, but **Instructor** is not part of the primary key (which is **Student_ID, Course_ID**).
- **3NF**: The table violates **3NF** because of the **transitive dependency**: **Instructor → Instructor_Phone** (non-prime attribute **Instructor_Phone** depends on non-prime attribute **Instructor**).
- **BCNF**: The table violates **BCNF** because **Instructor → Instructor_Phone**, and **Instructor** is not a superkey.

Step 3: Decompose into 3NF (or BCNF) Using Decomposition Rules

To bring the table to **3NF**, we'll break it down into smaller tables:

1. Student_Courses Table (Removes partial dependency):

Student_ID	Course_ID	Student_Name
101	301	Alice
102	301	Bob
101	302	Alice
103	303	Charlie

- **Primary key**: **Student_ID, Course_ID**
- **Functional dependency**: **Student_ID, Course_ID → Student_Name** (no partial dependencies)

2. Course_Instructors Table (Removes transitive dependency):

Course_ID	Instructor	Instructor_Phone
301	Dr. Smith	123-456-7890
302	Dr. Jones	987-654-3210
303	Dr. White	555-123-4567

- **Primary key**: **Course_ID**
- **Functional dependency**: **Course_ID → Instructor**, **Instructor → Instructor_Phone**

3. Instructor Table (Store information about instructors separately):

Instructor	Instructor_Phone
Dr. Smith	123-456-7890
Dr. Jones	987-654-3210
Dr. White	555-123-4567

- **Primary key**: **Instructor**
- **Functional dependency**: **Instructor → Instructor_Phone**

Step 4: Verify Decomposition

- **Lossless Join**: We can join the three tables back together on the appropriate keys (**Student_ID, Course_ID**, and **Course_ID**) to reconstruct the original table without losing any data.
- **Dependency Preservation**: The functional dependencies are preserved in the decomposed tables. For example, **Course_ID → Instructor** is directly represented in the **Course_Instructors Table**, and **Instructor → Instructor_Phone** is represented in the **Instructor Table**.
- **No Redundancy**: Each table now stores related information, and there is no unnecessary duplication of data. **Instructor_Phone** is stored only in the **Instructor Table**, and **Student_Name** is stored only in the **Student_Courses Table**.

Summary of the Decomposed Tables:

1. **Student_Courses Table**:

Student_ID	Course_ID	Student_Name
101	301	Alice
102	301	Bob
101	302	Alice
103	303	Charlie

2. **Course_Instructors Table**:

Course_ID	Instructor	Instructor_Phone
301	Dr. Smith	123-456-7890
302	Dr. Jones	987-654-3210
303	Dr. White	555-123-4567

3. **Instructor Table**:

Instructor	Instructor_Phone
Dr. Smith	123-456-7890
Dr. Jones	987-654-3210
Dr. White	555-123-4567

Conclusion:

This example shows the **decomposition** process to bring a table to **3NF** (or even **BCNF**). By splitting the data into smaller, more manageable tables and following the **decomposition rules** (lossless decomposition, dependency preservation, and key preservation), we eliminate redundancy and ensure the data is stored in a normalized manner. This decomposition also makes it easier to maintain and enforce consistency across the database.

In Database Management Systems (DBMS), **lossless decomposition** refers to the process of decomposing a relational schema (table) into smaller schemas (subtables) in such a way that no information is lost in the process. After decomposition, you should be able to reconstruct the original relation (table) from the decomposed relations using natural joins without any loss of data.

Lossless decomposition is crucial for **normalization**, a process that reduces redundancy and ensures that the database is free of anomalies. A decomposition is **lossless** if the natural join of the decomposed relations results in the original relation, and no data is lost or incorrectly discarded.

Formal Definition of Lossless Decomposition:

A decomposition of a relation **R** into two (or more) sub-relations **R1** and **R2** is **lossless** if:

1. **R1 ∩ R2 → R1** or
2. **R1 ∩ R2 → R2**.

In simple terms, for the decomposition to be lossless, the intersection of the decomposed relations must contain enough information (a **key**) to allow the original relation to be reconstructed by joining the decomposed relations.

Conditions for Lossless Decomposition:

A decomposition of a relation **R** into sub-relations **R1**, **R2**, ..., **Rn** is lossless if:

- There exists a set of attributes (called **common attributes**) between the decomposed relations such that the attributes from the original relation can be **reconstructed** using a **join**.
- The **common attributes** between the relations should include a **candidate key** from the original relation.

Example of Lossless Decomposition:

Step 1: Original Relation

Consider a relation **R** with the schema: R(A,B,C,D)R(A, B, C, D)R(A,B,C,D) This relation contains the following functional dependencies (FDs):

- **A → B**
- **C → D**

Step 2: Decompose the Relation

We want to decompose **R** into two sub-relations:

- **R1(A, B)**
- **R2(C, D)**

Step 3: Check for Lossless Decomposition

To check if this decomposition is lossless, we use the following approach:

- Find the common attributes between **R1** and **R2**. Here, there are no common attributes between **R1** and **R2**.
  This means we need to check if the decomposition satisfies the **lossless join condition**.
- **R1 ∩ R2** is the empty set **{}**.
- In this case, we check if the **common attributes** in **R1** and **R2** form a key in **R**. The common attribute is empty, so the decomposition is **not lossless**.

Example of Lossless Decomposition:

Let's consider another example where the decomposition is **lossless**.

Step 1: Original Relation

Consider a relation **R** with the schema: R(A,B,C)R(A, B, C)R(A,B,C) This relation has the following functional dependencies:

- **A → B**
- **B → C**

Step 2: Decompose the Relation

We want to decompose **R** into two sub-relations:

- **R1(A, B)**
- **R2(B, C)**

Step 3: Check for Lossless Decomposition

To check for lossless decomposition, look at the **common attribute** between **R1** and **R2**. In this case, the common attribute is **B**.

Now, check if **B** is a **key** in the original relation **R**:

- From **A → B**, we can deduce that **A** determines **B**, so **B** is not a key.
- From **B → C**, we can deduce that **B** determines **C**, so **B** determines both **C** and **A**.

Thus, **B** is a key for the original relation **R**.

Since the common attribute **B** is a key for the original relation, this decomposition is **lossless**.

Step 4: Reconstructing the Original Relation

To reconstruct the original relation, you would perform a **natural join** of the two decomposed relations: R1(A,B) and R2(B,C)R1(A, B) \ \text{and} \ R2(B, C)R1(A,B) and R2(B,C)

This will result in: R(A,B,C)R(A, B, C)R(A,B,C)

Lossless Decomposition in Normal Forms:

Lossless decomposition is critical in the normalization process, especially when converting a table into **3NF (Third Normal Form)** or **BCNF (Boyce-Codd Normal Form)**. When decomposing a table during normalization, ensuring losslessness guarantees that no information is lost, and that data can be reconstructed if needed.

Key Takeaways:

- **Lossless decomposition** ensures that no information is lost when decomposing a relation into smaller relations.

- A decomposition is lossless if the intersection of the decomposed relations contains enough attributes to recreate the original relation.
- Decompositions based on keys (such as in 3NF or BCNF) help achieve lossless decomposition and maintain data integrity.

Dependency-Preserving Decomposition in DBMS

In **Database Management Systems (DBMS)**, **dependency-preserving decomposition** refers to a process in which a relation (table) is decomposed into smaller relations (subtables), and all the original functional dependencies (FDs) are preserved in the decomposed relations. This means that after decomposition, we can enforce the original functional dependencies in the decomposed relations without needing to join them back together.

Why Dependency-Preserving Decomposition is Important:

- **Simplifies enforcement of constraints**: Dependency-preserving decomposition allows us to enforce constraints on individual relations without needing to perform complex joins.
- **Efficiency**: Since no additional joins are required to check dependencies, queries and operations on the database are more efficient.
- **Data integrity**: By preserving the original functional dependencies, we ensure that the decomposed relations still uphold the integrity rules of the original relation.

Conditions for Dependency-Preserving Decomposition:

A decomposition is dependency-preserving if, after decomposing a relation **R** into sub-relations **R1, R2, ..., Rn**, the union of the functional dependencies in **R1, R2, ..., Rn** is equivalent to the set of functional dependencies in **R** (i.e., the decomposition preserves all original dependencies).

However, it is important to note that **not all decompositions are dependency-preserving**. Sometimes, to achieve a lossless decomposition, we may have to sacrifice dependency preservation.

Example of Dependency-Preserving Decomposition:

Let's go through an example of a dependency-preserving decomposition.

Step 1: Original Relation

Consider a relation **R** with the schema: R(A,B,C,D)R(A, B, C, D)R(A,B,C,D) And the following functional dependencies:

- **A → B**
- **B → C**
- **A → D**

Step 2: Decompose the Relation

We want to decompose the relation **R** into two sub-relations:

- **R1(A, B, C)**
- **R2(A, D)**

Step 3: Check if the Decomposition is Dependency-Preserving

To check if this decomposition is dependency-preserving, we need to see if all the original functional dependencies are still enforceable within the decomposed relations **R1** and **R2**:

- **A → B**: This dependency is preserved in **R1** because **A** and **B** are both present in **R1**.
- **B → C**: This dependency is preserved in **R1** because **B** and **C** are present in **R1**.
- **A → D**: This dependency is preserved in **R2** because **A** and **D** are present in **R2**.

Thus, the original functional dependencies are all preserved in the decomposed relations. Therefore, the decomposition is **dependency-preserving**.

Step 4: Reconstructing the Original Relation

To reconstruct the original relation **R**, we would perform a **natural join** on **R1** and **R2**: R1(A,B,C) and R2(A,D)R1(A, B, C) \ \text{and} \ R2(A, D)R1(A,B,C) and R2(A,D)

This would yield: R(A,B,C,D)R(A, B, C, D)R(A,B,C,D)

Non-Dependency-Preserving Decomposition Example:

Step 1: Original Relation

Consider a relation **R** with the schema: R(A,B,C,D)R(A, B, C, D)R(A,B,C,D) And the following functional dependencies:

- **A → B**
- **B → C**
- **A → D**

Step 2: Decompose the Relation

Suppose we decompose **R** into:

- **R1(A, B, C)**
- **R2(B, D)**

Step 3: Check if the Decomposition is Dependency-Preserving

Now, let's check if the original functional dependencies are still enforceable in the decomposed relations:

- **A → B**: This dependency is preserved in **R1** because **A** and **B** are both in **R1**.
- **B → C**: This dependency is preserved in **R1** because **B** and **C** are both in **R1**.
- **A → D**: This dependency is not preserved in either **R1** or **R2**. **A** is in **R1**, but **D** is in **R2**, and there is no way to enforce this dependency directly within the decomposed relations.

This decomposition **does not** preserve the dependency **A → D**, so it is **not dependency-preserving**.

Dependency-Preserving Decomposition in Normalization:

- In the process of **normalization** (such as achieving **3NF** or **BCNF**), it is often desirable to have both **lossless** and **dependency-preserving** decompositions.
- **BCNF** decompositions, in particular, may sometimes result in a lossless but **non-dependency-preserving** decomposition. However, when we prioritize both losslessness and dependency preservation, it can become challenging to achieve both goals simultaneously.

Trade-off Between Lossless and Dependency-Preserving Decomposition:

- **Lossless decomposition**: Ensures that no information is lost, and the original relation can be reconstructed by joining the decomposed relations.
- **Dependency-preserving decomposition**: Ensures that all functional dependencies can be enforced directly in the decomposed relations without needing to join them back together.

It is possible to achieve one without the other. For example:

- A decomposition can be **lossless but not dependency-preserving** (in cases where dependencies are spread across different decomposed relations).
- A decomposition can be **dependency-preserving but not lossless** (though this is less common).

Conclusion:

Dependency-preserving decomposition ensures that all original functional dependencies are maintained in the decomposed relations, allowing for efficient enforcement of constraints without the need for joins. It is particularly useful in normalizing databases to reduce redundancy while maintaining data integrity. However, achieving both **losslessness** and **dependency preservation** in the same decomposition may not always be possible, and trade-offs may be required depending on the normalization goals.

Dependency in DBMS

What are Dependencies in DBMS?

- A dependency is a constraint that governs or defines the relationship between two or more attributes.
- In a database, it happens when information recorded in the same table uniquely determines other information stored in the same table.
- This may also be described as a relationship in which knowing the value of one attribute (or collection of attributes) in the same table tells you the value of another attribute (or set of attributes).
- It's critical to understand database dependencies since they serve as the foundation for database normalization.

What Normalization Stands for?

Normalization is a method of organizing data in a database that helps to reduce data redundancy, insertion, update, and deletion errors. It is the process of assessing relation schemas based on functional relationships and primary keys.

This allows you to limit the amount of space a database takes up while also ensuring that the data is kept correctly.

Normalization Need

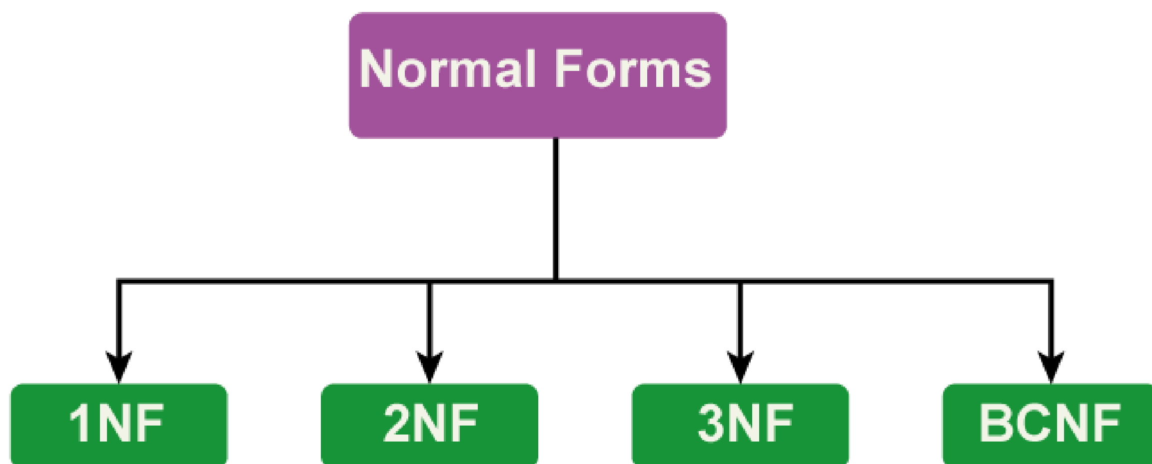
As previously stated, normalization is used to eliminate data redundancy. It offers a mechanism for removing the following anomalies from the database and making it more consistent:

A database anomaly is a fault in the database caused by insufficient preparation and redundancy.

- **Insertion anomalies** occur when we are unable to insert data into a database due to the absence of particular attributes at the moment of insertion.
- **Updation anomalies** occur when the same data items with the same values are repeated but are not related to each other.
- A **deletion anomaly** happens when deleting one part of the data results in the deletion of the other necessary information from the database.

Normal Forms

As illustrated in the image below, there are four types of normal forms that are commonly used in relational databases:



1. **First Normal Form (1NF):**
   A relation is in 1NF if all of its attributes are single-valued or if it lacks any multi-valued or composite attributes, i.e., every attribute is an atomic attribute. The 1NF is violated if there is a composite or multi-valued attribute. To resolve this, we can construct a new row for each of the multi-valued attribute values in order to transform the table into the 1NF.
2. **Second Normal Form (2NF):**
   Normalization of 1NF to 2NF relations entails the removal of incomplete dependencies. When any non-prime attributes, i.e., qualities that are not part of the candidate key, are not totally functionally reliant on one of the candidate keys, a partial dependency occurs. To be in second normal form, a relational table must obey the following rules:
   - The table must be presented in the first normal form.
   - It must not have any partial dependencies, which means that all non-prime attributes must be totally functionally dependent on the primary key.
3. **Third Normal Form (3NF):**
   Normalization of 2NF to 3NF relations entails the removal of transitive dependencies. To be in the third normal form, a relational table must obey the following rules:
   - The table should be in second normal form.
   - There are no non-prime attributes that are transitively dependent on the primary key.
   - At least one of the following conditions must be met for each functional dependency X -> Z:
   - The table's super key is X.
   - Z is a key feature of the table.
4. **Boyce-Codd Normal Form (BCNF):**
   Boyce-Codd Normal Form is a more advanced variant of 3NF since it has more limitations than 3NF.
   To be in Boyce-Codd normal form, a relational table must fulfil the following rules:
   - The table must be in the "Third Normal Form".
   - For every non-trivial functional dependency X -> Y, X is the table's superkey. That is, if Y is a prime attribute, X cannot be a non-prime attribute.

Types of Dependencies in DBMS
In DBMS, it has the following types:
- Functional Dependency
- Fully-Functional Dependency
- Transitive Dependency
- Multivalued Dependency
- Partial Dependency

Now, let's get started with Functional Dependency.
Functional Dependencies:
A functional dependency (FD) is a relationship that exists between two attributes in a database, typically the primary key and additional non-key attributes. Consider it a link between two qualities of the same relation.
A dependency is denoted by an arrow "→".

If Cdetermines Dfunctionally, thenC→D.
Functional dependency, indicated as C→ D, is a relationship between two sets of attributes, C and D. In this case, C is referred to as the "**determinant**", and Dis referred to as the "**dependent**".
Functional Dependency aids in the maintenance of database data quality.
Functional Dependency Rules:
Inference Rules

- **Axioms:**

A relational database's functional dependencies can be inferred using Armstrong's axioms, a set of inference principles. Armstrong, William W., created them.
Functional Dependencies Axioms:

1. **The reflexive rule** states that if D is a subset of C, then D is determined by C., i.e. C→D.
2. **The augmentation rule**, also known as the partial dependency rule, states that if D is determined by C, then CZ determines DZ for any Z.Every non-key attribute is required to be totally dependent on the Primary Key, according to it. i.e. If C→D, then CZ→DZ for any Z.
3. **Transitivity rule** states that if D is determined by C and Z is determined by D, then C must also determine Z., i.e. if C→D and D→Z, then C→Z.

- **Decomposition:**

It is a rule that stipulates that if a table appears to contain two entities determined by the same primary key, it should be split into two independent tables.
According to this rule, if C determines D and Z, C also determines D and Z individually.i.e. if C→DZ then C→D and C→Z.

- **Union**

It suggests that if two tables are independent yet have the same Primary Key, they should be combined.It states that C must determine D and Z if C determines D and C determines Z.
i.e. if C→D and C→Z then C→DZ.
Terms:
Dependent: It is shown on the functional dependency diagram's right side.
Determinant: It is shown on the functional dependency Diagram's left side.
Non-normalized table: A table containing redundant data.

Examples

Example 1: Here, we have a table named **Student**.

<Student>

StuID	StuName	StuAge
E01	Rose	14
E02	Rolly	13

Here, **StuName** in the preceding table is functionally dependent on **StuID** since **StuName** can only accept one value for the specified value of StuID, i.e. Because a student's name can be uniquely determined from an ID, StuName can be considered to be dependent on StuID.

1. StuID→StuName
   However, the converse assertion (StuName?>StuID) is false because multiple students can have the same name but have different StuID's.
   Example 2: We have a table **Employee.**
   <Employee>

Employee_No	E_Name	E_Salary	Address
1	Dolly	60000	Seoul
2	Flora	48000	BukchonHanok
3	Anni	35000	Seoul

We can Deduce Several Valid Functional Dependencies from the Preceding Table:
In this case, knowing the value of Employee_No allows us to access E_Name, Address, E_Salary, and so on. As a result, the Address, E Name, and E Salary are all functionally dependent on Employee No.

- Employee_No→ {E_Name, E_Salary, Address}: Employee _No can decide the values of fields E_Name, E_Salary, and Address in this case, resulting in a legal Functional dependence.
- Employee_No→E_Salary, Because Employee_No can determine the entire set of {E_Name, E_Salary, and Address}, it can also determine its subset E_Salary.
- More valid functional dependents include: Employee_No→name, {Employee_No, E_Name }→(E_Salary, Address}, and so on.

Here are Some Invalid Functional Dependencies:

- E_Name→E_Salary: This is not an acceptable functional dependencybecause employees with the same name can have different salaries.
- Address→E_Salary: Different salaries can be given to the employees of the same Address; for example, E_Salary 60000 and 35000 in the preceding table belong to employees of the same address, "Seoul"; hence Address→E_Salary is an incorrect functional dependency.
- More invalid functional dependencies include: E_Name→Employee_No, {E_Name, E_Salary}→Employee_No, and so on.

Types of Functional Dependencies

Functional Dependencies

Trivial Functional Dependency

Non- Trivial Functional Dependency

Multivalued Dependency

Transitive Dependency

1. Trivial Functional Dependency:
    1. A "dependent" in Trivial functional dependency is always a subset of the "determinant".
    2. A functional dependency is said to be trivial if the attributes on its right side are a subset of the attributes on its left side.
    3. If D is a subset of C, C→D is referred to as a Trivial Functional Dependency.

Example: Take a look at the Student table below.

<Student>

Roll_No	S_Name	S_Age
1	John	13
2	Riya	12
3	Giya	15
4	Jolly	16

- {Roll_No, S_ Name} →S_Name is a Trivial functional dependency in this case because the dependant S_Name is a subset of the determinant {Roll_No, S_Name}.
- { Roll_No } → { Roll_No }, { S_Name } → { S_Name } and { S_Age } → { S_Age } are also Trivial.

2. Non-Trivial Functional Dependency

- It is the inverse of Trivial functional dependence. Formally, a Dependent is a Non-Trivial functional dependency if it is not a subset of the determinant.
- If D is not a subset of C, C→D is said to have a non-trivial functional dependency. Non-trivial functional dependency is defined as a functional dependency C→ D where C is a set of attributes and D is also a set of attributes but not a subset of C.

Example: Consider the Student table below.

<Student>

Roll_No	S_Name	S_Age
1	John	13
2	Riya	12
3	Giya	15
4	Jolly	16

- Roll_No→S_Name is a non-trivial functional dependency in this case since S_Name(dependent) is not a subset of Roll_No (determinant).
- Similarly, {Roll_No, Name}→ Age are non-trivial functional dependencies.

3. Multivalued Functional Dependency

- In multivalued functional dependency, attributes in the dependent set are not dependent on one another.
- For example, C {D, Z}is referred to as a Multivalued functional dependency if there is no functional dependency between D and Z.

Example: Take a look at the **Student** table below.
<Student>

Roll_No	S_Name	S_Age
1	John	13
2	Riya	12
3	Giya	15
4	Jolly	16

- {Roll_No}→ {S_Name, S_Age) is a Multivalued functional dependency in this case because the "dependent values" S_ Name and S_Ageare not functionally dependent (i.e. S_Name→S_Ageor S_ Age→S_ Name does not exist).

4. Transitive Functional Dependency

- Consider two functional dependencies, C→ D and D→Z; C→Z must exist according to the transitivity principle. This is referred to as a **Transitive Functional dependency**.
- In transitive functional dependency, the dependent is dependent on the determinant indirectly.

Example: Consider the Student table below.
<Student>

Roll_No	S_Name	S_Department	Street_No
1	John	AC	12
2	Riya	BH	11
3	Giya	MV	14
4	Jolly	CD	18

- Roll_No→S_Department and S_Department→Street_No are correct here. As a result, Roll_No→Street_Number is a valid functional dependency, according to the principle of transitivity.

Benefits of Functional Dependency

- Functional Dependency prevents data duplication. As a result, the same data does not appear several times in that database.
- It assists you in maintaining the database's data quality.
- It assists you in defining database semantics and constraints.
- It aids you in spotting flawed designs.
- It aids you in locating database design information.
- The Normalization method begins with identifying the potential keys in the relation. It is impossible to locate candidate keys and normalize the database without functional dependencies.

Fully Functional Dependency

A functional dependency C→D,afully functional dependency is one in which, if any attribute x from C is removed, the "dependency" no longer exists.
If D is "fullyfunctional dependent" on C, it is not functionally dependent on any of the valid subsets of C.
i.e.Attribute Zin the relation CDE->Zis "fully functionally dependent"on CDEand not on any appropriate subset of CDE. That is, CDEsubsets such as CD, DE, C, D, and so on cannot determine Z.
Also;

- Full Functional Dependency corresponds to the Second Normal Form normalization standard.
- Functional dependency improves the data quality of our database.
- In this dependency, the non-prime property is functionally reliant on the candidate key.
- The full dependency on database attributes helps to assure data integrity and eliminate data abnormalities.

Example: Here, we have a table named **Supply**.

<Supply>

Seller_Id	Product_id	T_price
1	1	530
2	1	535
1	2	100
2	2	101
3	1	342

According to the Table, neither **Seller_id** nor **Product_id**can uniquely determine the price, but both **Seller_id**and**Product_id**combined can.
As a result, we can say that **T_price** is "fully functionally dependent" on **Seller_id** nor **Product_id**.
This outlines and demonstrates our fully functional dependency:

1. { Seller_id , Product_id } →T_Price

Partial Functional Dependency

A functional dependency C → D, If the dependency doeshold after removing any attribute x from C, then it is said to be a **Partial Functional Dependency**.
A functional dependency C→Y, If D is functionally dependent on C and may be determined by any appropriate subset of C, there is a partial dependency.
i.e. We have an CF->D, C->E, and E->D relation. Now, let us compute the closure of {C+}=CED. In this case, C can determine D on its own, implying that D is partially dependent on CF.

Also;

- In partial functional dependency, the non-prime attribute is functionally dependent on a component of a candidate key.
- The normalizing standard of the Second Normal Form does not apply to Partial Functional Dependency. 2NF, on the other hand, eliminates Partial Dependency.
- Partially dependent data does not improve data quality. It must be removed before normalization in the second normal form may occur.

Cause of Partial Dependency:

Partial dependency happens when a non-prime attribute is functionally dependent on a portion of the given candidate key, as we saw in the preceding section.
In other words, partial dependency arises when an attribute in a table is dependent on only a portion of the primary key rather than the entire key.
Example: We have a table called **Student** here.

<Student>

Roll_No	S_Name	S_Course
1	John	DBMS
2	Riya	C++
3	Giya	Java
4	Jolly	C

We can see that the attributes S_Name and Roll_No can both uniquely identify a S_Course. As a result, we might argue that the relationship is partly dependent.

Transitive Dependency

A transitive dependence is any non-prime attribute other than the candidate key that is reliant on another non-prime attribute that is wholly dependent on the candidate key.

Transitive Dependency occurs when an indirect interaction results in functional dependency. As a result, if C→ D and D ->Z are true, then C ->Z is a transitive dependency.

Transitive dependency causes deletion, update, and insertion errors in the database and is regarded as poor database design.

To reach 3NF, one must first eliminate Transitive Dependency.

Note:

Only when two Functional Dependencies establish an indirect functional dependency can it be transitive. As an example,

When the following functional dependencies hold true, C →E is a transitive dependency:

- C ->D
- D does not imply C
- C→E

Only in the case of some given relation of three or more attributes can transitive dependency occur effortlessly. Such a dependency aids us in normalizing the database in its 3rd Normal Form (3NF).

Example: Here, we have a table **Telecast_show**.

<Telecast_show>

Id_show	Id_telecast	Type_telecast	Cost_CD
F01	S01	Romantic	30
F02	S02	Thriller	50
F03	S03	Comedy	20

(Because of a transitive functional relationship, the table above is not in its 3NF.)

Id_show→Id_telecast

Id_telecast→Type_telecast

As a result, the following functional dependency is transitive.

1. Id_show→Type_telecast
   Avoiding Transitive Functional Dependency

According to the preceding statement, the relation <Telecast> violates the 3NF (3rd Normal Form). To address this violation, we must split the tables in order to remove the transitive functional relationship.

<show>

Id_show	Id_telecast	Cost_CD
F01	S01	30
F02	S02	50
F03	S03	20

<telecast>

Id_telecast	Type_telecast
S09	Thriller
S05	Romantic
S09	Comedy

The preceding relationship is now in the Third Normal Form (3NF) of Normalization.

Multivalued Dependency

The termMultivalued Dependency refers to having several rows in a particular table. As a result, it implies that there are multiple other rows in the same table. A multivalued dependency would thus preclude the 4NF. Any multivalued dependency would involve at least three table attributes.

When two separate attributes in a given table are independent of each other, multivalued dependency occurs. However, both of these are dependent on a third factor. At least two of the attributes are reliant on the third attribute in the multivalued dependence. This is why it always includes at least three of the qualities.

Example: Here we have a table **Car.**

<Car>

Model_car	Month_manu	Col_or
S2001	Jan	Yellow
S2002	Feb	Red
S2003	March	Yellow
S2004	April	Red
S2005	May	Yellow
S2006	June	Red

In this scenario, the columns Col_or and Month_manu are both dependent on Model-car but independently of one another. As a result, we can refer to both of these columns as multivalued. Thus, they are dependent on Model_car. Here is a diagram of the dependencies we covered earlier:

1. Model_car → →Month_manu
2. Model_car → → Col_or

Why Do We Use Multivalued Dependency in DBMS?

When we face these two different ways, we always employ multivalued conditions:

- When we wish to test the relationships or determine whether they are legal under certain arrangements of practical and multivalued dependencies.
- When we want to know what restrictions exist on the arrangement of legal relationships; as a result, we will only be concerned with the relationships that fulfil a specific arrangement of practical and multivalued dependencies.

Occurrence:

- When two qualities in a table are independent of each other yet reliant on a third property, this is referred to as multivalued dependence.
- Because multivalued Dependency requires a minimum of two variables that are independent of each other in order to be dependent on the third variable, the minimum number of variables necessary is two.

DBMS Dependency Conditions for Multivalued Dependency:

If all of the following conditions are met, we can state that multivalued dependency exists.

If any attribute 'C' has many dependencies on 'D,' for any relation R, for all the pair data values in table row R1 and table row R2, such that the relation

1. R1[C]=R2[C]

exists, and there is a relationship between row R3 and row R4 in the table such that

1. R1[C] = R2[C] = R3[C] = R4[C]
2. R1[D] = R3[D], R2[D] = R4[D]

   Then we can assert the existence of Multivalued Dependency (MVD).
   That is, in Rows R1, R2, R3, and R4,
   R1[C], R2[C], R3[C], and R4[C] must all have the same value.
   The value of R1[D] should be equal to R3[D], and the value of R2[D] should be equal to R4[D].
   Example: Here, we have a table **Course.**
   <Course>

Row_	Name_	Course_work_	Hobby_
R1	Ronit	Java	Dancing
R2	Ronit	Python	Singing
R3	Ronit	Java	Dancing
R4	Ronit	Python	Singing

Because we have distinct values of Course_work_ and Hobby_ for the same value of Name "Ronit," we have multivalued dependents on Name_.
Verificationof <Course> table.
Let us now examine the condition of MVD(Multivalued Dependency) in our table.
Condition 1:
R1[C] = R2[C] = R3[C] = R4[C]
From the table;
R1[C] = R2[C] = R3[C] = R4[C] = 'Ronit'.
As a result, condition 1 appears to be met.
Condition 2:
R1[D] = R3[D], R2[D] = R4[D]
From the table;
R1[D] = R3[D] = 'Java', R2[D] = R4[D]= 'Python'.
As a result, condition 2 appears to be met as well.
Condition 3:
R1[e] = R4[e], R2[e] = R3[e]
We can draw a conclusion from the table.
R1[E] = R4[E] = 'Dancing', R2[E] = R3[E] = 'Singing'.
As a result, condition 3 is likewise satisfied, indicating that MVD occurs in the provided situation.
We have now;

1. C →→ D

   And from the table, we obtained the following;
   Name_ →→Course_work_
   And for C →→ E, we have
   Name_ → Hobby_
   Finally, in the given table, we can conclude with the conditional relation as

1. Name_ →→Course_work_
2. Name_ →→ Hobby_

Conclusion:

- The functional dependency of a relation defines how its attributes are related to one another. It aids in the preservation of data quality in the database. It is represented by an arrow "→".
- C→D represents the functional dependency of C on D. In 1974, William Armstrong proposed a few axioms or laws about functional dependency. They have The Reflexivity Rule, Augmentation Rule, and Transitivity Rule.
- Functional dependencies are classified into four categories. Functional dependency can be classified as trivial, non-trivial, multivalued, or transitive.
- Functional dependencies have various benefits, including keeping the database design clean, clarifying the meaning and limits of the databases, and eliminating data redundancy.
- In a database, a transitive dependency is an indirect relationship between items in the same table that results in a functional dependency.
- A transitive dependency, by definition, requires three or more properties.
- To meet the "Third Normal Form (3NF) " normalization standard, any transitive dependency must be removed.
- Transitive dependency causes deletion, update, and insertion errors in the database and is regarded as poor database design.
- When the values of two independent attributes, say D and E, are determined by the third attribute C, multivalued dependence exists.
- The symbol for Multivalued Dependency is 'C--> D'.
- As a result, we can state that in order for a multivalued dependency to exist in a relation R.
- Two components of a single property, say B and C, should be mutually independent of each other.
- For two tuples of R, say C and D, the full attributes of C may have distinct values for component D.
- Similarly, For two tuples of R, say C and E, the component E may have distinct values for the full attributes of C.
- When an attribute in a database depends solely on a portion of the candidate key rather than the entire key, this is referred to as partial functional dependence.i.e. "Prime → Non-Prime".
- Normal forms are used to eliminate redundancy and minimize database storage.
- In 1NF, we check the atomicity of a relation's characteristics.
- We look for partial dependencies in a relation using 2NF.
- We look for transitive dependencies in a relation using 3NF.
- BCNF looks for superkeys in the LHS of all functional dependents.