**A Functional Dependency**

A functional dependency is a constraint that specifies the relationship between two sets of attributes where one set can accurately determine the value of other sets. It is denoted as **X → Y**, where X is a set of attributes that is capable of determining the value of Y. The attribute set on the left side of the arrow, **X**is called **Determinant**, while on the right side, **Y**is called the **Dependent**. Functional dependencies are used to mathematically express relations among database entities and are very important to understand advanced concepts in Relational Database System and understanding problems in competitive exams like Gate.

**Example:**

|  |  |  |  |
| --- | --- | --- | --- |
| roll\_no | **name** | **dept\_name** | **dept\_building** |
| 42 | abc | CO | A4 |
| 43 | pqr | IT | A3 |
| 44 | xyz | CO | A4 |
| 45 | xyz | IT | A3 |
| 46 | mno | EC | B2 |
| 47 | jkl | ME | B2 |

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

* roll\_no → { name, dept\_name, dept\_building },→  Here, roll\_no can determine values of fields name, dept\_name and dept\_building, hence a valid Functional dependency
* roll\_no → dept\_name , Since, roll\_no can determine whole set of {name, dept\_name, dept\_building}, it can determine its subset dept\_name also.
* dept\_name → dept\_building ,  Dept\_name can identify the dept\_building accurately, since departments with different dept\_name will also have a different dept\_building
* More valid functional dependencies: roll\_no → name, {roll\_no, name} ⇢ {dept\_name, dept\_building}, etc.

**Here are some invalid functional dependencies:**

* name → dept\_name   Students with the same name can have different dept\_name, hence this is not a valid functional dependency.
* dept\_building → dept\_name    There can be multiple departments in the same building, For example, in the above table departments ME and EC are in the same building B2, hence dept\_building → dept\_name is an invalid functional dependency.
* More invalid functional dependencies: name → roll\_no, {name, dept\_name} → roll\_no, dept\_building → roll\_no, etc.

**Armstrong’s axioms/properties of functional dependencies:**

1. **Reflexivity:**If Y is a subset of X, then X→Y holds by reflexivity rule  
   For example, {roll\_no, name} → name is valid.
2. **Augmentation:** If X → Y is a valid dependency, then XZ → YZ is also valid by the augmentation rule.  
   For example, If {roll\_no, name} → dept\_building is valid, hence {roll\_no, name, dept\_name} → {dept\_building, dept\_name} is also valid.→
3. **Transitivity**: If X → Y and Y → Z are both valid dependencies, then X→Z is also valid by the Transitivity rule.  
   For example, roll\_no → dept\_name & dept\_name → dept\_building, then roll\_no → dept\_building is also valid.

**Types of Functional dependencies in DBMS:**

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

**1. Trivial Functional Dependency**

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

**For example,**

|  |  |  |
| --- | --- | --- |
| roll\_no | name | age |
| 42 | abc | 17 |
| 43 | pqr | 18 |
| 44 | xyz | 18 |

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

**2. Non-trivial Functional Dependency**

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

**For example,**

|  |  |  |
| --- | --- | --- |
| roll\_no | name | age |
| 42 | abc | 17 |
| 43 | pqr | 18 |
| 44 | xyz | 18 |

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

**3. Multivalued Functional Dependency**

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

**For example,**

|  |  |  |
| --- | --- | --- |
| roll\_no | name | age |
| 42 | abc | 17 |
| 43 | pqr | 18 |
| 44 | xyz | 18 |
| 45 | abc | 19 |

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

**4. Transitive Functional Dependency**

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

**For example,**

|  |  |  |  |
| --- | --- | --- | --- |
| enrol\_no | name | dept | building\_no |
| 42 | abc | CO | 4 |
| 43 | pqr | EC | 2 |
| 44 | xyz | IT | 1 |
| 45 | abc | EC | 2 |

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

**Decomposition**

**Decomposition in DBMS is to break a relation into multiple relations to bring it into an appropriate normal form.**It helps to remove redundancy, inconsistencies, and anomalies from a database.

## Introduction

If a relation is not properly decomposed, then it may lead to other problems like information loss, etc. There are **two types of decomposition** as shown below:

## Rules for Decomposition

Whenever we decompose a relation, there are certain properties that must be satisfied to ensure no information is lost while decomposing the relations. These properties are:

1. Lossless Join Decomposition.
2. Dependency Preserving.

## Lossless Join Decomposition

**A lossless Join decomposition ensures two things:**

* No information is lost while decomposing from the original relation.
* If we join back the sub decomposed relations, the same relation that was decomposed is obtained.

We can follow certain rules to ensure that the decomposition is a lossless join decomposition Let’s say we have a relation R and we decomposed it into R1 and R2, then the rules are:

1. The union of attributes of both the sub relations R1 and R2 must contain all the attributes of original relation R.

**R1 ∪ R2 = R**

1. The intersection of attributes of both the sub relations R1 and R2 must not be null, i.e., there should be some attributes that are present in both R1 and R2.

**R1 ∩ R2 ≠ ∅**

1. The intersection of attributes of both the sub relations R1 and R2 must be the superkey of R1 or R2, or both R1 and R2.

**R1 ∩ R2 = Super key of R1 or R2**

Let’s see an example of a lossless join decomposition. Suppose we have the following relation **EmployeeProjectDetail** as:

**<EmployeeProjectDetail>**

| **Employee\_Code** | **Employee\_Name** | **Employee\_Email** | **Project\_Name** | **Project\_ID** |
| --- | --- | --- | --- | --- |
| 101 | John | john@demo.com | Project103 | P03 |
| 101 | John | john@demo.com | Project101 | P01 |
| 102 | Ryan | ryan@example.com | Project102 | P02 |
| 103 | Stephanie | stephanie@abc.com | Project102 | P02 |

Now, we decompose this relation into EmployeeProject and ProjectDetail relations as:

**<EmployeeProject>**

| **Employee\_Code** | **Project\_ID** | **Employee\_Name** | **Employee\_Email** |
| --- | --- | --- | --- |
| 101 | P03 | John | john@demo.com |
| 101 | P01 | John | john@demo.com |
| 102 | P04 | Ryan | ryan@example.com |
| 103 | P02 | Stephanie | stephanie@abc.com |

The primary key of the above relation is {Employee\_Code, Project\_ID}.

**<ProjectDetail>**

| **Project\_ID** | **Project\_Name** |
| --- | --- |
| P03 | Project103 |
| P01 | Project101 |
| P04 | Project104 |
| P02 | Project102 |

The primary key of the above relation is {Project\_ID}.

Now, let’s see if this is a lossless join decomposition by evaluating the rules discussed above:

**Let’s first check the EmployeeProject ∪ ProjectDetail:**

**<EmployeeProject ∪ ProjectDetail>**

| **Employee\_Code** | **Project\_ID** | **Employee\_Name** | **Employee\_Email** | **Project\_Name** |
| --- | --- | --- | --- | --- |
| 101 | P03 | John | john@demo.com | Project103 |
| 101 | P01 | John | john@demo.com | Project101 |
| 102 | P04 | Ryan | ryan@example.com | Project104 |
| 103 | P02 | Stephanie | stephanie@abc.com | Project102 |

As we can see all the attributes of **EmployeeProject** and **ProjectDetai**l are in **EmployeeProject** **∪ ProjectDetail** relation and it is the same as the original relation. So the first condition holds.

**Now let’s check the EmployeeProject ∩ ProjectDetail:**

**<EmployeeProject ∩ ProjectDetail>**

| **Project\_ID** |
| --- |
| P03 |
| P01 |
| P04 |
| P02 |

As we can see this is not null, so the the second condition holds as well. Also **the EmployeeProject ∩ ProjectDetail = Project\_Id**. This is the super key of the ProjectDetail relation, so the third condition holds as well.

Now, since **all three conditions hold** for our decomposition, this is a **lossless join decomposition.**

## Lossless vs Lossy Decomposition

In a lossy decomposition, one or more of these conditions would fail and we will not be able to recover Complete information as present in the original relation. For example, let's say we decompose our original relation EmployeeProjectDetail into EmployeeProject and ProjectDetail relations as:

**<EmployeeProject>**

| **Employee\_Code** | **Employee\_Name** | **Employee\_Email** |
| --- | --- | --- |
| 101 | John | john@demo.com |
| 102 | Ryan | ryan@example.com |
| 103 | Stephanie | stephanie@abc.com |

The primary key of the above relation is {Employee\_Code}.

**<ProjectDetail>**

| **Project\_ID** | **Project\_Name** |
| --- | --- |
| P03 | Project103 |
| P01 | Project101 |
| P04 | Project104 |
| P02 | Project102 |

The primary key of the above relation is **{Project\_ID}.**

Now, the intersection **EmployeeProject ∩ ProjectDetail** is null. Therefore there is no way for us to map a project to its employees. Thus this is a **lossy decomposition**.

## Dependency Preserving

The second property of lossless decomposition is dependency preservation which says that after decomposing a relation R into R1 and R2, all dependencies of the original relation R must be present either in R1 or R2 or they must be derivable using the combination of functional dependencies present in R1 and R2.

Let’s understand this from the same example above:

**<EmployeeProjectDetail>**

| **Employee\_Code** | **Employee\_Name** | **Employee\_Email** | **Project\_Name** | **Project\_ID** |
| --- | --- | --- | --- | --- |
| 101 | John | john@demo.com | Project103 | P03 |
| 101 | John | john@demo.com | Project101 | P01 |
| 102 | Ryan | ryan@example.com | Project104 | P04 |
| 103 | Stephanie | stephanie@abc.com | Project102 | P02 |

In this relation we have the following FDs:

* Employee\_Code -> {Employee\_Name, Employee\_Email}
* Project\_ID - > Project\_Name

Now, after decomposing the relation into EmployeeProject and ProjectDetail as:

**<EmployeeProject>**

| **Employee\_Code** | **Project\_ID** | **Employee\_Name** | **Employee\_Email** |
| --- | --- | --- | --- |
| 101 | P03 | John | john@demo.com |
| 101 | P01 | John | john@demo.com |
| 102 | P04 | Ryan | ryan@example.com |
| 103 | P02 | Stephanie | stephanie@abc.com |

In this relation we have the following FDs:

* **Employee\_Code -> {Employee\_Name, Employee\_Email}**

**<ProjectDetail>**

| **Project\_ID** | **Project\_Name** |
| --- | --- |
| P03 | Project103 |
| P01 | Project101 |
| P04 | Project104 |
| P02 | Project102 |

In this relation we have the following FDs:

* **Project\_ID - > Project\_Name**

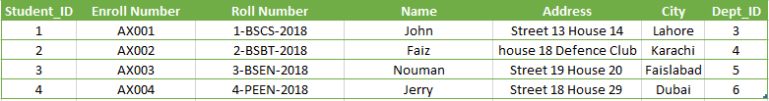
As we can see that all FDs in EmployeeProjectDetail are either part of the EmployeeProject or the ProjectDetail, So this decomposition is **dependency preserving**.

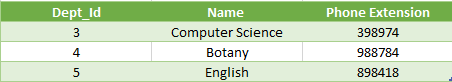
**Anamoly**

Like [Redundancy](https://bitalksbi.com/redundancy/) anomalies are also very important topic for exams if you’re a student, and for interviews if you are looking for a job . Following are the ones we should be concerned about .

1. Insert Anomaly
2. Delete Anomaly
3. Update Anomaly

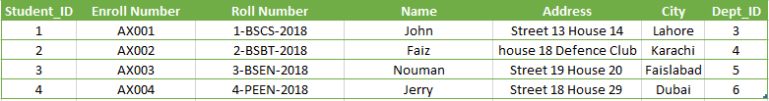
## INSERT Anomaly in Database

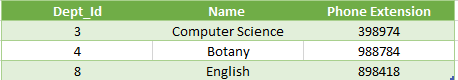




An Insert Anomaly occurs when attributes cannot be inserted into the database without the presence of other attributes. Usually when a child is inserted without parent.  
  
Jerry is a new Student with department id 6. There is no Department with this Dept\_ID 6. Hence , the anomaly. The usual behaviour should be a new department id with 6 and only then Student could have it.

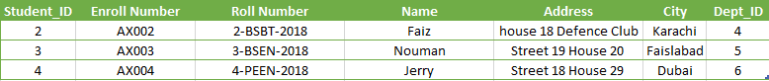
## UPDATE Anomaly in Database





When duplicated data is updated at one instance and not across all instances where it was duplicated. That’s an update anomaly .  See below English department has now Dept\_ID 8 , but unfortunately it was not updated in Student table.

## DELETE Anomaly in Database



Anomalies in DBMS

Now if someone decides to delete Computer Science department , he may end up deleting all student’s data who had the department of Computer Science. So to say deletion of some attribute which causes deletion of other attributes is deletion anomaly.

# Normalization

A large database defined as a single relation may result in data duplication. This repetition of data may result in:

* Making relations very large.
* It isn't easy to maintain and update data as it would involve searching many records in relation.
* Wastage and poor utilization of disk space and resources.
* The likelihood of errors and inconsistencies increases.

So to handle these problems, we should analyze and decompose the relations with redundant data into smaller, simpler, and well-structured relations that are satisfy desirable properties. Normalization is a process of decomposing the relations into relations with fewer attributes.

## What is Normalization?

* 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 also used to eliminate undesirable characteristics like Insertion, Update, and Deletion Anomalies.
* Normalization divides the larger table into smaller and links them using relationships.
* The normal form is used to reduce redundancy from the database table.

**Why do we need Normalization?**

The main reason for normalizing the relations is removing these anomalies. Failure to eliminate anomalies leads to data redundancy and can cause data integrity and other problems as the database grows. Normalization consists of a series of guidelines that helps to guide you in creating a good database structure.

**Data modification anomalies can be categorized into three types:**

* **Insertion Anomaly:** Insertion Anomaly refers to when one cannot insert a new tuple into a relationship due to lack of data.
* **Deletion Anomaly:** The delete anomaly refers to the situation where the deletion of data results in the unintended loss of some other important data.
* **Updatation Anomaly:** The update anomaly is when an update of a single data value requires multiple rows of data to be updated.

## Types of Normal Forms:

Normalization works through a series of stages called Normal forms. The normal forms apply to individual relations. The relation is said to be in particular normal form if it satisfies constraints.

**Following are the various types of Normal forms:**



|  |  |
| --- | --- |
| **Normal Form** | **Description** |
| [1NF](https://www.javatpoint.com/dbms-first-normal-form) | A relation is in 1NF if it contains an atomic value. |
| [2NF](https://www.javatpoint.com/dbms-second-normal-form) | A relation will be in 2NF if it is in 1NF and all non-key attributes are fully functional dependent on the primary key. |
| [3NF](https://www.javatpoint.com/dbms-third-normal-form) | A relation will be in 3NF if it is in 2NF and no transition dependency exists. |
| BCNF | A stronger definition of 3NF is known as Boyce Codd's normal form. |
| [4NF](https://www.javatpoint.com/dbms-forth-normal-form) | A relation will be in 4NF if it is in Boyce Codd's normal form and has no multi-valued dependency. |
| [5NF](https://www.javatpoint.com/dbms-fifth-normal-form) | A relation is in 5NF. If it is in 4NF and does not contain any join dependency, joining should be lossless. |

## Advantages of Normalization

* Normalization helps to minimize data redundancy.
* Greater overall database organization.
* Data consistency within the database.
* Much more flexible database design.
* Enforces the concept of relational integrity.

## Disadvantages of Normalization

* You cannot start building the database before knowing what the user needs.
* The performance degrades when normalizing the relations to higher normal forms, i.e., 4NF, 5NF.
* It is very time-consuming and difficult to normalize relations of a higher degree.
* Careless decomposition may lead to a bad database design, leading to serious problems.

# First Normal Form (1NF)

* A relation will be 1NF if it contains an atomic value.
* It states that an attribute of a table cannot hold multiple values. It must hold only single-valued attribute.
* First normal form disallows the multi-valued attribute, composite attribute, and their combinations.

**Example:** Relation EMPLOYEE is not in 1NF because of multi-valued attribute EMP\_PHONE.

**EMPLOYEE table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_PHONE** | **EMP\_STATE** |
| 14 | John | 7272826385, 9064738238 | UP |
| 20 | Harry | 8574783832 | Bihar |
| 12 | Sam | 7390372389, 8589830302 | Punjab |

The decomposition of the EMPLOYEE table into 1NF has been shown below:

|  |  |  |  |
| --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_PHONE** | **EMP\_STATE** |
| 14 | John | 7272826385 | UP |
| 14 | John | 9064738238 | UP |
| 20 | Harry | 8574783832 | Bihar |
| 12 | Sam | 7390372389 | Punjab |
| 12 | Sam | 8589830302 | Punjab |

# Second Normal Form (2NF)

* In the 2NF, relational must be in 1NF.
* In the second normal form, all non-key attributes are fully functional dependent on the primary key

**Example:** Let's assume, a school can store the data of teachers and the subjects they teach. In a school, a teacher can teach more than one subject.

**TEACHER table**

|  |  |  |
| --- | --- | --- |
| **TEACHER\_ID** | **SUBJECT** | **TEACHER\_AGE** |
| 25 | Chemistry | 30 |
| 25 | Biology | 30 |
| 47 | English | 35 |
| 83 | Math | 38 |
| 83 | Computer | 38 |

In the given table, non-prime attribute TEACHER\_AGE is dependent on TEACHER\_ID which is a proper subset of a candidate key. That's why it violates the rule for 2NF.

To convert the given table into 2NF, we decompose it into two tables:

3

**TEACHER\_DETAIL table:**

|  |  |
| --- | --- |
| **TEACHER\_ID** | **TEACHER\_AGE** |
| 25 | 30 |
| 47 | 35 |
| 83 | 38 |

**TEACHER\_SUBJECT table:**

|  |  |
| --- | --- |
| **TEACHER\_ID** | **SUBJECT** |
| 25 | Chemistry |
| 25 | Biology |
| 47 | English |
| 83 | Math |
| 83 | Computer |

# Third Normal Form (3NF)

* A relation will be in 3NF if it is in 2NF and not contain any transitive partial dependency.
* 3NF is used to reduce the data duplication. It is also used to achieve the data integrity.
* If there is no transitive dependency for non-prime attributes, then the relation must be in third normal form.

A relation is in third normal form if it holds atleast one of the following conditions for every non-trivial function dependency X → Y.

1. X is a super key.
2. Y is a prime attribute, i.e., each element of Y is part of some candidate key.

**Example:**

**EMPLOYEE\_DETAIL table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_ZIP** | **EMP\_STATE** | **EMP\_CITY** |
| 222 | Harry | 201010 | UP | Noida |
| 333 | Stephan | 02228 | US | Boston |
| 444 | Lan | 60007 | US | Chicago |
| 555 | Katharine | 06389 | UK | Norwich |
| 666 | John | 462007 | MP | Bhopal |

**Super key in the table above:**

* 1. {EMP\_ID}, {EMP\_ID, EMP\_NAME}, {EMP\_ID, EMP\_NAME, EMP\_ZIP}....so on

**Candidate key:** {EMP\_ID}

**Non-prime attributes:** In the given table, all attributes except EMP\_ID are non-prime.

Here, EMP\_STATE & EMP\_CITY dependent on EMP\_ZIP and EMP\_ZIP dependent on EMP\_ID. The non-prime attributes (EMP\_STATE, EMP\_CITY) transitively dependent on super key(EMP\_ID). It violates the rule of third normal form.

That's why we need to move the EMP\_CITY and EMP\_STATE to the new <EMPLOYEE\_ZIP> table, with EMP\_ZIP as a Primary key.

**EMPLOYEE table:**

|  |  |  |
| --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_ZIP** |
| 222 | Harry | 201010 |
| 333 | Stephan | 02228 |
| 444 | Lan | 60007 |
| 555 | Katharine | 06389 |
| 666 | John | 462007 |

**EMPLOYEE\_ZIP table:**

|  |  |  |
| --- | --- | --- |
| **EMP\_ZIP** | **EMP\_STATE** | **EMP\_CITY** |
| 201010 | UP | Noida |
| 02228 | US | Boston |
| 60007 | US | Chicago |
| 06389 | UK | Norwich |
| 462007 | MP | Bhopal |
|  |  |  |

# Boyce Codd normal form (BCNF)

* BCNF is the advance version of 3NF. It is stricter than 3NF.
* A table is in BCNF if every functional dependency X → Y, X is the super key of the table.
* For BCNF, the table should be in 3NF, and for every FD, LHS is super key.

**Example:** Let's assume there is a company where employees work in more than one department.

**EMPLOYEE table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_COUNTRY** | **EMP\_DEPT** | **DEPT\_TYPE** | **EMP\_DEPT\_NO** |
| 264 | India | Designing | D394 | 283 |
| 264 | India | Testing | D394 | 300 |
| 364 | UK | Stores | D283 | 232 |
| 364 | UK | Developing | D283 | 549 |

**In the above table Functional dependencies are as follows:**

1. EMP\_ID  →  EMP\_COUNTRY
2. EMP\_DEPT  →   {DEPT\_TYPE, EMP\_DEPT\_NO}

**Candidate key: {EMP-ID, EMP-DEPT}**

The table is not in BCNF because neither EMP\_DEPT nor EMP\_ID alone are keys.

To convert the given table into BCNF, we decompose it into three tables:

**EMP\_COUNTRY table:**

|  |  |
| --- | --- |
| **EMP\_ID** | **EMP\_COUNTRY** |
| 264 | India |
| 264 | India |

**EMP\_DEPT table:**

|  |  |  |
| --- | --- | --- |
| **EMP\_DEPT** | **DEPT\_TYPE** | **EMP\_DEPT\_NO** |
| Designing | D394 | 283 |
| Testing | D394 | 300 |
| Stores | D283 | 232 |
| Developing | D283 | 549 |

**EMP\_DEPT\_MAPPING table:**

|  |  |
| --- | --- |
| **EMP\_ID** | **EMP\_DEPT** |
| D394 | 283 |
| D394 | 300 |
| D283 | 232 |
| D283 | 549 |

**Functional dependencies:**

1. EMP\_ID   →    EMP\_COUNTRY
2. EMP\_DEPT   →   {DEPT\_TYPE, EMP\_DEPT\_NO}

**Candidate keys:**

**For the first table:** EMP\_ID  
**For the second table:** EMP\_DEPT  
**For the third table:** {EMP\_ID, EMP\_DEPT}