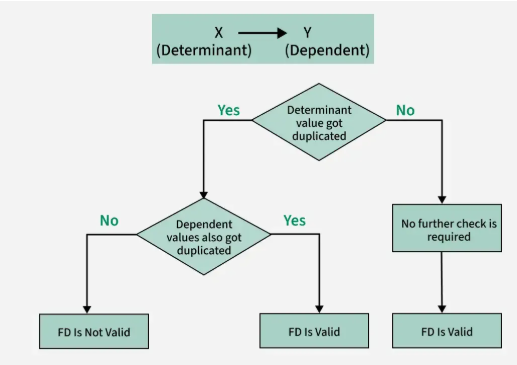
**1.Define functional dependency. Explain different types of Armstrong's axioms in functional dependency.**

Functional dependency in DBMS is an important concept that describes the relationship between attributes (columns) in a table. It helps in maintaining data integrity. And also It shows that the value of one attribute determines the other. Functional dependencies help maintain the quality of data in the database.

Suppose we have a student table with attributes: Stu\_Id, Stu\_Name, Stu\_Age. Here Stu\_Id attribute uniquely identifies the Stu\_Name attribute of student table because if we know the student id we can tell the student name associated with it. This is known as functional dependency and can be written as Stu\_Id→Stu\_Name or in words we can say Stu\_Name is functionally dependent on Stu\_Id.

Formally: If column A of a table uniquely identifies the column B of same table then it can represented as A->B (Attribute B is functionally dependent on attribute A).



**How to represent functional dependency in DBMS?**

Functional dependency is expressed in the form of equations. For example, if we have an employee record with fields "EmployeeID", "FirstName" and "LastName" we can specify the function as follows:

EmployeeID -> FirstName, LastName

To represent functional dependency in DBMS has two main features: left (LHS) and right (RHS) of the arrow (->).

For example, if we have a table with attributes "X", "Y" and "Z" and the attribute "X" can determine the value of the attributes "Y" and "Z".

X -> Y, Z

This symbol indicates that the value in property "X" determines the values ​​in property "Y" and "Z". So if you know the value of "X", you can also determine the value of "X" and "Z".

## Armstrong Axioms

The term Armstrong Axioms refers to the sound and complete set of inference rules or axioms, introduced by William W. Armstrong, that is used to test the logical implication of **functional dependencies**. If F is a set of functional dependencies then the closure of F, denoted as F+, is the set of all functional dependencies logically implied by F. Armstrong’s Axioms are a set of rules, that when applied repeatedly, generates a closure of functional dependencies.

### Axioms

* **Axiom of Reflexivity:**If A is a set of attributes and B is a subset of A, then A holds B. If B⊆A then A**→**B. This property is trivial property.
* **Axiom of Augmentation:**If **A→B**holds and Y is the attribute set, then **AY→BY** also holds. That is adding attributes to dependencies, does not change the basic dependencies. If **A→B**, then **AC→BC** for any C.
* **Axiom of Transitivity:**Same as the transitive rule in algebra, if **A→B** holds and **B→C** holds, then**A→C** also holds. **A→B** is called A functionally which determines B. If **X→Y** and**Y→Z**, then **X→Z.**

### **Secondary Rules**

These rules can be derived from the above axioms.

* **Union:**If **A→B** holds and **A→C** holds, then **A→BC** holds. If **X→Y** and **X→Z** then**X→YZ**.
* **Composition:**If**A→B**and **X→Y**hold, then **AX→BY** holds.
* **Decomposition:**If **A→BC** holds then **A→B**and **A→C** hold. If **X→YZ** then **X→Y** and **X→Z**.
* **Pseudo Transitivity:**If **A→B** holds and **BC→D**holds, then **AC→D** holds. If **X→Y** and **YZ→W** then **XZ→W**.
* **Self Determination:** It is similar to the Axiom of Reflexivity, i.e. **A→A**for any A.
* **Extensivity:** Extensivity is a case of augmentation. If **AC→A,**and **A→B**, then **AC→B**. Similarly, **AC→ABC** and **ABC→BC**. This leads to **AC→BC**.

## Advantages of Using Armstrong’s Axioms in Functional Dependency

* They provide a systematic and efficient method for inferring additional functional dependencies from a given set of functional dependencies, which can help to optimize [database design](https://www.geeksforgeeks.org/significance-of-database-design/).
* They can be used to identify redundant functional dependencies, which can help to eliminate unnecessary data and improve database performance.
* They can be used to verify whether a set of functional dependencies is a minimal cover, which is a set of dependencies that cannot be further reduced

without losing information.

## Disadvantages of Using Armstrong’s Axioms in Functional Dependency

* The process of using Armstrong’s axioms to infer additional functional dependencies can be computationally expensive, especially for large databases with many tables and relationships.
* The axioms do not take into account the semantic meaning of data, and may not always accurately reflect the relationships between data elements.
* The axioms can result in a large number of inferred functional dependencies, which can be difficult to manage and maintain over time.

**2. Discuss the following functional dependency.**

**i) Trivial FD ii) Fully functional dependency iii) MVD**

A functional dependency occurs when one attribute uniquely determines another attribute within a relation. It is a constraint that describes how attributes in a table relate to each other. If attribute A functionally determines attribute B we write this as the A→B.

Functional dependencies are used to mathematically express relations among database entities and are very important to understanding advanced concepts in Relational Database Systems.

## ****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.

Symbolically: A→B is trivial functional dependency if B is a subset of A.

The following dependencies are also trivial: A→A & B→B

**Example 1**:

* ABC -> AB
* ABC -> A
* ABC -> ABC

**Example 2:**

| **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.

**Example 1 :**

* Id -> Name
* Name -> DOB

**Example 2:**

| **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.

**Example:**

| **bike\_model** | **manuf\_year** | **Color** |
| --- | --- | --- |
| **tu1001** | 2007 | Black |
| **tu1001** | 2007 | Red |
| **tu2012** | 2008 | Black |
| **tu2012** | 2008 | Red |
| **tu2222** | 2009 | Black |
| **tu2222** | 2009 | Red |

In this table:

* **X**: bike\_model
* **Y**: color
* **Z**: manuf\_year

For each bike model (bike\_model):

1. There is a group of colors (color) and a group of manufacturing years (manuf\_year).
2. The colors do not depend on the manufacturing year, and the manufacturing year does not depend on the colors. They are independent.
3. The sets of color and manuf\_year are linked only to bike\_model.

That’s what makes it a multivalued dependency.

In this case these two columns are said to be multivalued dependent on bike\_model. These dependencies can be represented like this:

Read more about [Multivalued Dependency in DBMS](https://www.geeksforgeeks.org/multivalued-dependency-mvd-in-dbms/).

### **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.

**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.

**3.Define normalization? Explain about normalization process.**

**5. Explain about INF and 2NF with an example.**

**7. Discuss 4 NF and 5 NF with example.**

Normalization is the process of organizing the data and the attributes of a database. It is performed to reduce the data redundancy in a database and to ensure that data is stored logically. [Data redundancy in DBMS](https://www.scaler.com/topics/redundancy-in-dbms/) means having the same data but at multiple places. It is necessary to remove data redundancy because it causes anomalies in a database which makes it very hard for a database administrator to maintain it.

## Why Do We Need Normalization?

Normalization is used to reduce data redundancy. It provides a method to remove the following anomalies from the database and bring it to a more consistent state:

A database anomaly is a flaw in the database that occurs because of poor planning and redundancy.

1. **Insertion anomalies**: This occurs when we are not able to insert data into a database because some attributes may be missing at the time of insertion.
2. **Updation anomalies:** This occurs when the same data items are repeated with the same values and are not linked to each other.
3. **Deletion anomalies:** This occurs when deleting one part of the data deletes the other necessary information from the database.

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:

**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 |

## Difference Between 3NF and BCNF in DBMS

Here are the differences between 3NF and BCNF in DBMS:

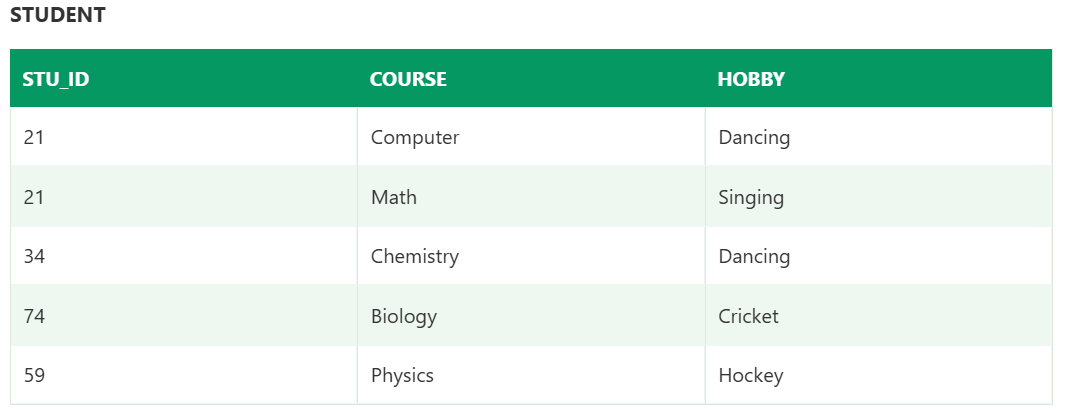
|  |  |  |
| --- | --- | --- |
| **Parameters** | **3NF** | **BCNF** |
| Strength | 3NF is comparatively less strong than that of the BCNF. | BCNF is comparatively much stronger than that of the 3NF. |
| Functional Dependencies | The functional dependencies in 3NF already exist in 2NF and INF. | The functional dependencies in BCNF already exist in 3NF, 2NF, and INF. |
| Redundancy | 3NF has a comparatively much higher redundancy. | BCNF has a comparatively much lower redundancy. |
| Functional Dependencies | In the case of 3NF, preservation occurs for all the functional dependencies. | In the case of BCNF, there is no preservation for all the functional dependencies. |
| Lossless Decomposition | Lossless decomposition is comparatively much easier to achieve in the case of 3NF. | Lossless decomposition is comparatively much harder to achieve in the case of BCNF. |

# Fourth normal form (4NF)

* A relation will be in 4NF if it is in Boyce Codd normal form and has no multi-valued dependency.
* For a dependency A → B, if for a single value of A, multiple values of B exists, then the relation will be a multi-valued dependency.

### Example

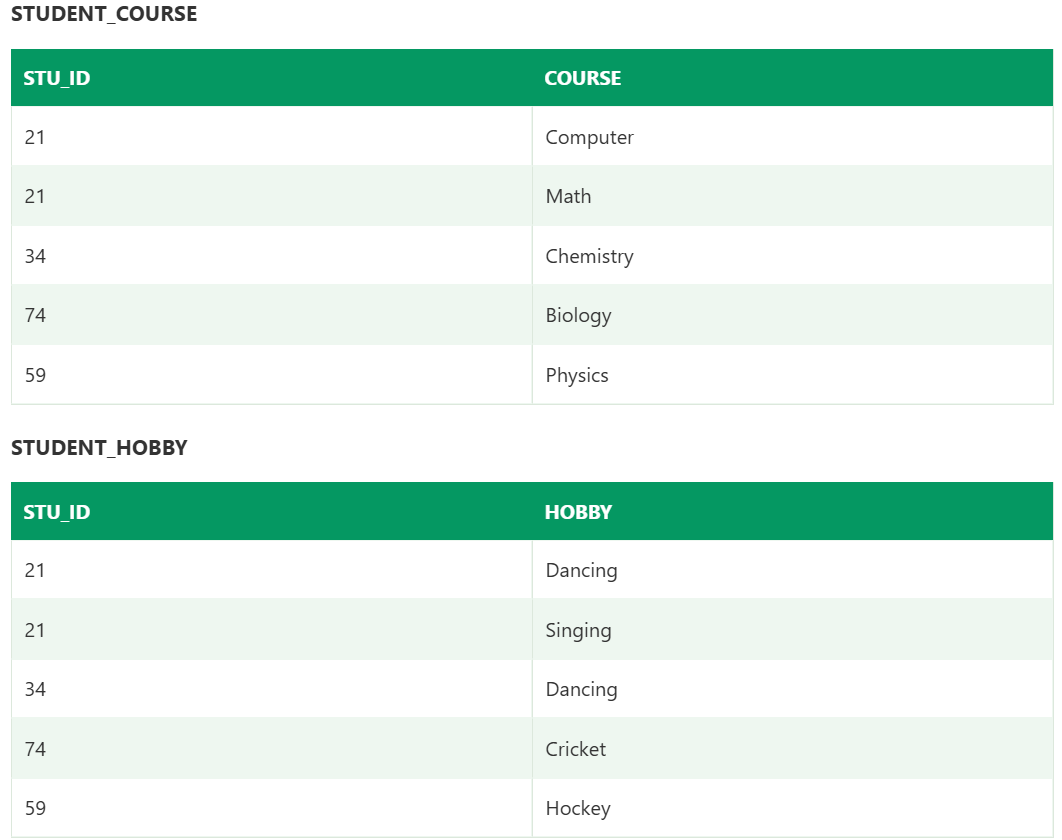
**STUDENT**



The given STUDENT table is in 3NF, but the COURSE and HOBBY are two independent entity. Hence, there is no relationship between COURSE and HOBBY.

In the STUDENT relation, a student with STU\_ID, **21** contains two courses, **Computer** and **Math** and two hobbies, **Dancing** and **Singing**. So there is a Multi-valued dependency on STU\_ID, which leads to unnecessary repetition of data.

So to make the above table into 4NF, we can decompose it into two tables:



# Fifth normal form (5NF)

* A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless.
* 5NF is satisfied when all the tables are broken into as many tables as possible in order to avoid redundancy.
* 5NF is also known as Project-join normal form (PJ/NF).

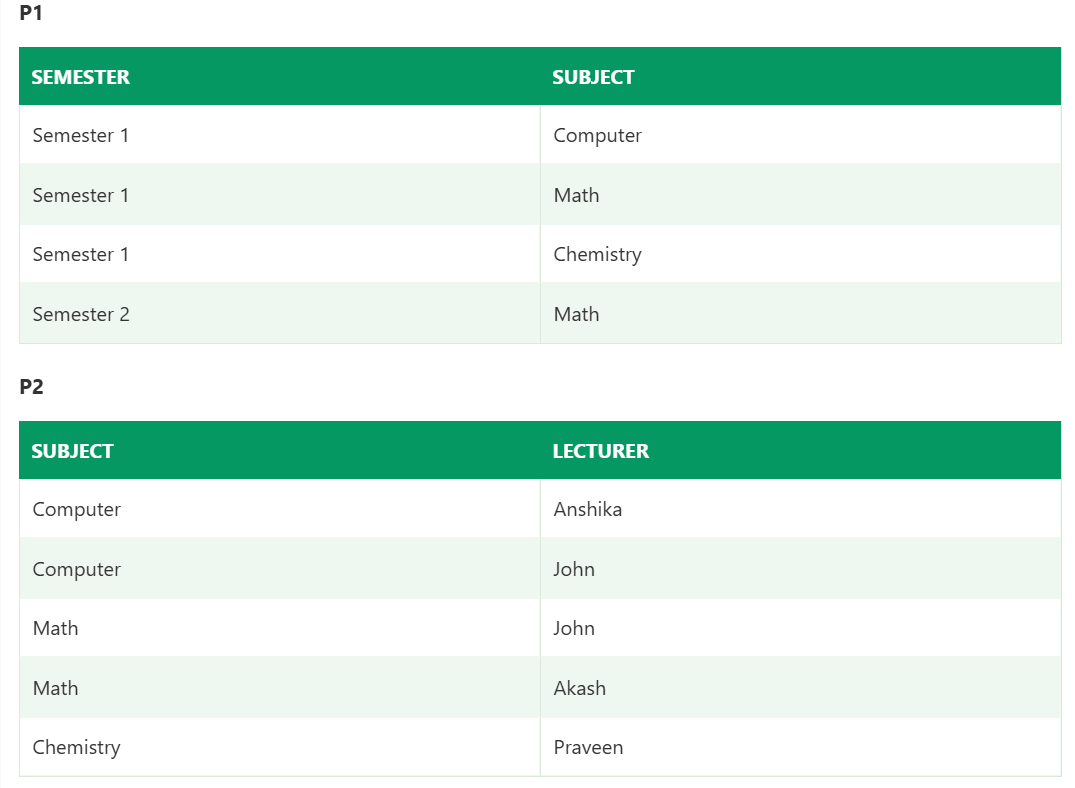
### Example

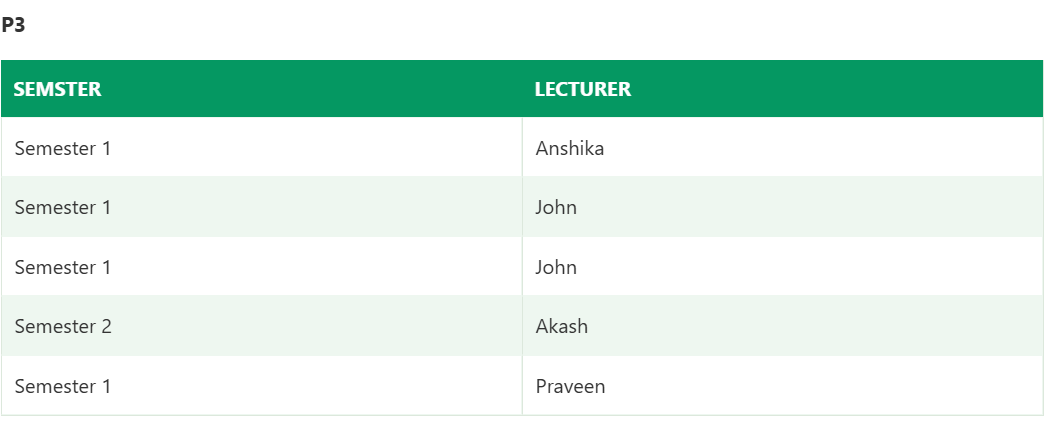


In the above table, John takes both Computer and Math class for Semester 1 but he doesn't take Math class for Semester 2. In this case, combination of all these fields required to identify a valid data.

Suppose we add a new Semester as Semester 3 but do not know about the subject and who will be taking that subject so we leave Lecturer and Subject as NULL. But all three columns together acts as a primary key, so we can't leave other two columns blank.

So to make the above table into 5NF, we can decompose it into three relations P1, P2 & P3:

****



**4.What are multi valued dependencies? Explain.**

**Multivalued dependency (MVD)** is a type of dependency that exists when a table contains more than one multivalued attribute and changes to one attribute can affect another attribute. In other words, MVD occurs when a table has a non-trivial relationship between attributes that are not part of the same composite key.

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

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



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

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

*BIKE\_MODEL -> -> MANUF\_YEAR*

*BIKE\_MODEL -> -> COLOR*

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

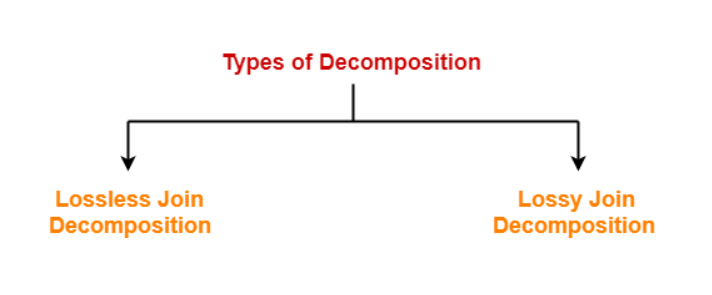
**6. Write Short Notes on Lossy join and Loss less-join.**

# Decomposition

# When a relation in the relational model is not in appropriate normal form then the decomposition of a relation is required.

* In a database, it breaks the table into multiple tables.
* If the relation has no proper decomposition, then it may lead to problems like loss of information.
* Decomposition is used to eliminate some of the problems of bad design like anomalies, inconsistencies, and redundancy.

## Types of Decomposition



### **Lossless Decomposition**

* If the information is not lost from the relation that is decomposed, then the decomposition will be lossless.
* The lossless decomposition guarantees that the join of relations will result in the same relation as it was decomposed.
* The relation is said to be lossless decomposition if natural joins of all the decomposition give the original relation.

**Example:**

**EMPLOYEE\_DEPARTMENT table:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_AGE** | **EMP\_CITY** | **DEPT\_ID** | **DEPT\_NAME** |
| 22 | Denim | 28 | Mumbai | 827 | Sales |
| 33 | Alina | 25 | Delhi | 438 | Marketing |
| 46 | Stephan | 30 | Bangalore | 869 | Finance |
| 52 | Katherine | 36 | Mumbai | 575 | Production |
| 60 | Jack | 40 | Noida | 678 | Testing |

The above relation is decomposed into two relations EMPLOYEE and DEPARTMENT

**EMPLOYEE table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_AGE** | **EMP\_CITY** |
| 22 | Denim | 28 | Mumbai |
| 33 | Alina | 25 | Delhi |
| 46 | Stephan | 30 | Bangalore |
| 52 | Katherine | 36 | Mumbai |
| 60 | Jack | 40 | Noida |

**DEPARTMENT table**

|  |  |  |
| --- | --- | --- |
| **DEPT\_ID** | **EMP\_ID** | **DEPT\_NAME** |
| 827 | 22 | Sales |
| 438 | 33 | Marketing |
| 869 | 46 | Finance |
| 575 | 52 | Production |
| 678 | 60 | Testing |

Now, when these two relations are joined on the common column "EMP\_ID", then the resultant relation will look like:

**Employee ⋈ Department**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_AGE** | **EMP\_CITY** | **DEPT\_ID** | **DEPT\_NAME** |
| 22 | Denim | 28 | Mumbai | 827 | Sales |
| 33 | Alina | 25 | Delhi | 438 | Marketing |
| 46 | Stephan | 30 | Bangalore | 869 | Finance |
| 52 | Katherine | 36 | Mumbai | 575 | Production |
| 60 | Jack | 40 | Noida | 678 | Testing |

Hence, the decomposition is Lossless join decomposition.

Example 2:

### Example of LossLess decomposition

**StudentCourse Table:**

Student\_Id Student\_Name Course\_Id Course\_Detail

---------- ------------- --------- -------------

S101 Chaitanya C01 Maths

S102 Ajeet C01 Maths

S103 Rahul C02 Science

S104 Steve C02 Science

S105 John C03 English

S101 Chaitanya C03 English

S102 Ajeet C02 Science

The primary key of given relation is {Student\_Id, Course\_Id}

This table has **redundant data** as the Course\_Id and Course\_Detail are common for several students. Let’s decompose this relation into two relations.

**Student Table:**  
The primary key of this table is {Student\_Id, Course\_Id}

Student\_Id Student\_Name Course\_Id

---------- ------------ ---------

S101 Chaitanya C01

S102 Ajeet C01

S103 Rahul C02

S104 Steve C02

S105 John C03

S101 Chaitanya C03

S102 Ajeet C02

**Course Table:**  
The primary key of this table is {Course\_Id}

Course\_Id Course\_Detail

--------- -------------

C01 Maths

C02 Science

C03 English

Let’s check all the three rules of lossless decomposition to check whether this decomposition is lossless or not.  
**Rule 1:**

{Student} U {Course}

**Union Result:**

Student\_Id Student\_Name Course\_Id Course\_Detail

---------- ------------- --------- -------------

S101 Chaitanya C01 Maths

S102 Ajeet C01 Maths

S103 Rahul C02 Science

S104 Steve C02 Science

S105 John C03 English

S101 Chaitanya C03 English

S102 Ajeet C02 Science

The union results in the original relation StudentCourse so we can say that the first rule holds true.

**Rule 2 & 3:**

R1 ∩ R2

**Result:**

Course\_Id

C01

C02

C03

**The result is not null so rule 2 holds true.**

The **result is a super key of the second relation R2** so the third rule also applies here.

**Rule 4: Dependencies in original relation:**

Student\_Id -> {Student\_Name}

Course\_Id -> {Course\_Detail}

These dependencies are still present in the decomposed relations. Thus we can say that this decomposition is dependency preserving.

Since all the three rules applies here, **the decomposition of relation StudentCourse into Student and Course is a lossless decomposition.**

### **2. Lossy Decomposition**

As the name suggests, in **lossy decomposition**, the information is lost during decomposition. The three rules that we discussed above would not apply in lossy decomposition. In lossy decomposition, one or more rules will fail.

Let’s take the same example that we discussed above.  
**StudentCourse Table:**

Student\_Id Student\_Name Course\_Id Course\_Detail

---------- ------------- --------- -------------

S101 Chaitanya C01 Maths

S102 Ajeet C01 Maths

S103 Rahul C02 Science

S104 Steve C02 Science

S105 John C03 English

S101 Chaitanya C03 English

S102 Ajeet C02 Science

**Now if we divide this relation like this:**  
**Student Table:**  
The primary key of this table is {Student\_Id}

Student\_Id Student\_Name

S101 Chaitanya

S102 Ajeet

S103 Rahul

S104 Steve

S105 John

**Course Table:**  
The primary key of this table is {Course\_Id}

Course\_Id Course\_Detail

C01 Maths

C02 Science

C03 English

This is a **lossy decomposition** as the intersection of Student and Course relation will return null so the second and third rule of lossless decomposition will fail here.

In this decomposition, the **relation of Student and Course is lost**, there is no way to form the original relation from these two relations as the information that suggests who is attending which course is lost during decomposition.

### Example 2

Let's consider a table of relations.

**<Employee\_Info>**

| **Emp\_ID** | **Emp\_Name** | **Emp\_Age** | **Emp\_Location** | **Dep\_ID** | **Dep\_Name** |
| --- | --- | --- | --- | --- | --- |
| EX001 | John | 30 | Chicago | Dep1 | Marketing |
| EX002 | Sam | 35 | New York | Dep2 | Sales |
| EX003 | Emily | 28 | Los Angeles | Dep3 | HR |

Now, let's decompose this table into two tables as follows:

**<Emp\_Details>**

| **Emp\_ID** | **Emp\_Name** | **Emp\_Age** | **Emp\_Location** |
| --- | --- | --- | --- |
| EX001 | John | 30 | Chicago |
| EX002 | Sam | 35 | New York |
| EX003 | Emily | 28 | Los Angeles |

**<Dep\_Details>**

| **Dep\_ID** | **Dep\_Name** |
| --- | --- |
| Dep1 | Marketing |
| Dep2 | Sales |
| Dep3 | HR |

If we attempt to join the two tables above, we will not be able to do so because the relation Emp\_ID is not part of the Dep\_Details relation.

Hence, this is an example of a lossy decomposition.

### Dependency Preserving

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

**9. Illustrate dependency preservation property of decomposition with an example.**

A **Database Management System (DBMS)**, **dependency-preserving decomposition** refers to the process of breaking down a complex database schema into simpler, smaller tables, such that all the functional dependencies of the original schema are still enforceable without needing to perform additional joins.

This approach is crucial for database normalization as it minimizes redundancy, prevents anomalies, and improves the efficiency of database queries. To achieve dependency-preserving decomposition, algorithms like **lossless join decomposition** and **dependency-preserving decomposition** are applied, ensuring that all original dependencies can be represented directly in the decomposed tables.

**Example:**

Suppose R is a relational schema and F is the set of functional dependencies on R. If R is decomposed into relations R1, R2, ………….…… Rn , each holding functional dependencies F1, F2, …….……… Fn respectively. We can say, F` = F1 U F2 U ………..… U Fn.

The key concepts of dependency-preserving decomposition include:

* **Functional Dependency Preservation:** This means that after decomposition, the functional dependencies in the original schema must still hold true in the decomposed schema.
* **Lossless Join Property:** The decomposition must allow for the original relation to be reconstructed from the decomposed relations without any data loss, ensuring no information is discarded.
* **Normalization:** The decomposition often aims to normalize the schema to higher normal forms (like 3NF or BCNF), which further eliminates redundancy and dependency anomalies.
* **Minimal Redundancy:** By ensuring the decomposition preserves [functional dependencies](https://www.geeksforgeeks.org/what-is-functional-dependency-in-dbms/), it minimizes data redundancy and helps in avoiding data anomalies.

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

### **Solution:**

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

*Let us find closure of F1 and F2  
To find closure of F1, consider all combination of ABC. i.e., find closure of A, B, C, AB, BC and AC  
Note ABC is not considered as it is always ABC*

*closure(A) = { A } // Trivial  
closure(B) = { B } // Trivial  
closure(C) = {C, A, D} but D can’t be in closure as D is not present R1.  
= {C, A}  
C–> A // Removing C from right side as it is trivial attribute*

*closure(AB) = {A, B, C, D}  
= {A, B, C}  
AB –> C // Removing AB from right side as these are trivial attributes*

*closure(BC) = {B, C, D, A}  
= {A, B, C}  
BC –> A // Removing BC from right side as these are trivial attributes*

*closure(AC) = {A, C, D}  
NULL SET*

*F1 {C–> A, AB –> C, BC –> A}.  
Similarly F2 { C–> D }*

*In the original Relation Dependency { AB –> C , C –> D , D –> A}.  
AB –> C is present in F1.  
C –> D is present in F2.  
D –> A is not preserved.*

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

Dependency-preserving decomposition enhances database efficiency by:

* **Eliminating Redundancy:** It helps reduce unnecessary repetition of data, leading to smaller storage requirements.
* **Maintaining Integrity:** By preserving functional dependencies, the database ensures consistent data with fewer chances of anomalies like update, insert, or delete anomalies.
* **Improving Query Performance:** With a well-decomposed schema, it’s easier to optimize queries as the smaller tables are often faster to process.
* **Simplifying Updates:** Since data is more normalized, updates become simpler and more efficient, reducing the risk of inconsistencies.

I**mp Note:** The 1NF, 2NF, and 3NF are valid for dependency-preserving decomposition.

## Step-by-Step Approach to Dependency Preserving Decomposition in DBMS

* In this technique, the original relation is decomposed into smaller relations in such a way that the resulting relations preserve the functional dependencies of the original relation. This is important because if the decomposition results in losing any of the original functional dependencies, it can lead to data inconsistencies and anomalies.
* To achieve dependency preserving decomposition, there are various algorithms available, such as the Boyce-Codd Normal Form ([BCNF](https://www.geeksforgeeks.org/boyce-codd-normal-form-bcnf/)) decomposition and the Third Normal Form (3NF) decomposition. These algorithms are based on the concept of functional dependencies and are used to identify the attributes that should be grouped together to form smaller relations.
* The BCNF decomposition algorithm is used to decompose a relation into smaller relations in such a way that each resulting relation is in BCNF. BCNF is a higher normal form than 3NF and is used when there are multiple candidate keys in a relation.
* The [3NF decomposition](https://www.geeksforgeeks.org/third-normal-form-3nf/) algorithm is used to decompose a relation into smaller relations in such a way that each resulting relation is in 3NF. 3NF is a normal form that ensures that there are no transitive dependencies between the attributes of a relation.
* Overall, dependency preserving decomposition is an important technique in [DBMS](https://www.geeksforgeeks.org/introduction-of-dbms-database-management-system-set-1/) for improving database efficiency while maintaining data consistency and integrity. It is important to choose the right decomposition algorithm based on the specific requirements of the database to achieve the desired results.

## 10. Discuss the concept of surrogate key with example.

A key is a column, or group of columns, in a database management system (DBMS) that uniquely identifies every row in a table. Natural keys and surrogate keys are the two categories of keys.

* **Natural Key:** A column, or group of columns, that is generated from the table’s data is known as a natural key. For instance, since it uniquely identifies every client in the table, the customer ID column in a customer table serves as a natural key.
* **Surrogate key**: A column that is not generated from the data in the database is known as a surrogate key. Rather, the DBMS generates a unique identifier for you. In database tables, surrogate keys are frequently utilized as primary keys.

## Surrogate Key

A surrogate key also called a synthetic primary key, is generated when a new record is inserted into a table automatically by a database that can be declared as the primary key of that table. It is the sequential number outside of the database that is made available to the user and the application or it acts as an object that is present in the database but is not visible to the user or application.

We can say that, in case we do not have a natural primary key in a table, then we need to artificially create one in order to uniquely identify a row in the table, this key is called the surrogate key or synthetic primary key of the table. However, the surrogate key is not always the primary key. Suppose we have multiple objects in a database that are connected to the surrogate key, then we will have a many-to-one association between the primary keys and the surrogate key and the surrogate key cannot be used as the primary key.

## ****Features of the Surrogate Key****

* It is automatically generated by the system.
* It holds an anonymous integer.
* It contains a unique value for all records of the table.
* The value can never be modified by the user or application.
* The surrogate key is called the factless key as it is added just for our ease of identification of unique values and contains no relevant fact(or information) that is useful for the table.

**Consider an example:**  
Suppose we have two tables of two different schools having the same column registration\_no, name, and percentage, each table having its own natural primary key, that is registration\_no.

Table of school A:

| **registration\_no** | **Name** | **percentage** |
| --- | --- | --- |
| 210101 | Harry | 90 |
| 210102 | Maxwell | 65 |
| 210103 | Lee | 87 |
| 210104 | Chris | 76 |

Table of school B:

| **registration\_no** | **name** | **percentage** |
| --- | --- | --- |
| CS107 | Taylor | 49 |
| CS108 | Simon | 86 |
| CS109 | Sam | 96 |
| CS110 | Andy | 58 |

Now, suppose we want to merge the details of both the schools in a single table.  
Resulting table will be:

| **surr\_no** | **registration\_no** | **name** | **percentage** |
| --- | --- | --- | --- |
| 1 | 210101 | Harry | 90 |
| 2 | 210102 | Maxwell | 65 |
| 3 | 210103 | Lee | 87 |
| 4 | 210104 | Chris | 76 |
| 5 | CS107 | Taylor | 49 |
| 6 | CS108 | Simon | 86 |
| 7 | CS109 | Sam | 96 |
| 8 | CS110 | Andy | 58 |

 As we can observe the above table and see that registration\_no cannot be the primary key of the table as it does not match with all the records of the table though it is holding all unique values of the table . Now , in this case, we have to artificially create one primary key for this table. We can do this by adding a column surr\_no in the table that contains anonymous integers and has no direct relation with other columns . This additional column of surr\_no is the surrogate key of the table.

## Why use Surrogate Key in DBMS?

There are several reasons to use surrogate keys in database tables:

1. **Uniqueness:** Data integrity is improved by the guaranteed uniqueness of surrogate keys.
2. **Stability:** Since surrogate keys do not depend on any business rules or data value, they have a lower chance of changing over time.
3. **Efficiency:** Compared to natural keys, surrogate keys are frequently smaller and process more quickly.
4. **Flexibility:** In the event that the natural key changes, rows can still be uniquely identified using surrogate keys.

## ****Advantages of the Surrogate Key****

* As there is no direct information related with the table, so the changes are only based on the requirements of the application.
* Performance is enhanced as the value of the key is relatively smaller.
* The key value is guaranteed to contain unique information .
* As it holds smaller constant values , this makes integration of the table easy.
* Enables us to run fast queries (as compared to the natural[primary key](https://www.geeksforgeeks.org/primary-key-in-dbms/))

## ****Disadvantages of the Surrogate Key****

* The surrogate key value can never be used as a search key.
* As the key value has no relation to the data of the table, so third normal form is violated.
* The extra column for surrogate key will require extra disk space.
* We will need extra IO when we have to insert or update data of the table.

## 8. Apply how 3NF rules will predict the relation into a BCNF with examples.

To move a relation from 3NF to Boyce-Codd Normal Form (BCNF), ensure that for every non-trivial functional dependency X → Y, the attribute X (the determinant) is a superkey (or candidate key) of the relation. If a relation is not in BCNF, decompose it based on the functional dependencies that violate the BCNF condition.

Here's a breakdown with an example:

Understanding 3NF and BCNF

* **3NF:**

In 3NF, all non-key attributes are fully functionally dependent on the primary key, and there are no transitive dependencies.

* **BCNF:**

BCNF is a stricter form of normalization than 3NF. It requires that every determinant in a functional dependency be a superkey (or candidate key).

Example

Let's consider a relation R(A, B, C, D) with the following functional dependencies:

* A → B
* A, C → D
* B → C

1. Check for 3NF

* A is the primary key.
* B is fully dependent on A.
* C is fully dependent on A, B.
* D is fully dependent on A, C.
* This relation is in 3NF because all non-key attributes are dependent on the primary key, and there are no transitive dependencies.

2. Check for BCNF

* A → B: A is a superkey.
* A, C → D: A, C is a superkey.
* B → C: B is not a superkey. This violates BCNF.

3. Decompose the Relation

Since B → C violates BCNF, we need to decompose the relation.

* **New Relation 1:** R1(A, B, C) with functional dependency A → B and B → C.
* **New Relation 2:** R2(A, C, D) with functional dependency A, C → D.

4. Verify BCNF in Decomposed Relations

* **R1(A, B, C):**
  + A → B: A is a superkey.
  + B → C: B is a superkey.
* **R2(A, C, D):**
  + A, C → D: A, C is a superkey.

After decomposition, both R1 and R2 are in BCNF because every determinant in a functional dependency is a superkey.