

UNIT-3

Introduction to SQL : Select Queries, Constraints: Data Manipulation Language –Insert ,Delete, Update, form of basic SQL query , UNION, INTERSECT, and EXCEPT, Nested Queries, Co-related Queries aggregation operators, NULL values, complex integrity constraints in SQL.

Concept of Joins: Join, Outer Join , Left Outer Join, Right Outer Join, Self Join

Schema Refinement : Problems caused by redundancy, decompositions, problems related to decomposition, reasoning about functional dependencies, FIRST, SECOND, THIRD normal forms, BCNF, lossless join decomposition, multi-valued dependencies, FOURTH normal form, FIFTH normal form.

What is SQL?

SQL is the standard language for dealing with Relational Databases. SQL can be used to insert, search, update, and delete database records. SQL can do lots of other operations, including optimizing and maintenance of databases.

SQL Full Form

SQL stands for Structured Query language, pronounced as "S-Q-L" or sometimes as "See-Quel"... Relational databases like MySQL Database, Oracle, MS SQL Server, Sybase, etc. use ANSI SQL.

How to Use SQL

SQL Code Example:

```
SELECT * FROM Members WHERE Age > 30 ;
```

SQL syntaxes used in different databases are almost similar, though few RDBMS use a few different commands and even proprietary SQL syntaxes.

Types of SQL Statements

Here are five types of widely used SQL queries.

- Data Definition Language (DDL)
- Data Manipulation Language (DML)
- Data Control Language (DCL)
- Transaction Control Language (TCL)
- Data Query Language (DQL)

List of SQL Commands

Here's a list of some of the most commonly used **SQL commands**:

- **CREATE** - defines the database structure schema
- **INSERT** - inserts data into the row of a table
- **UPDATE** - updates data in a database
- **DELETE** - removes one or more rows from a table
- **SELECT** - selects the attribute based on the condition described by the WHERE clause
- **DROP** - removes tables and databases

SQL Language elements

Here are important elements of SQL language:

- **Keywords:** Each SQL statement contains single or multiple keywords.
- **Identifiers:** Identifiers are names of objects in the database, like user IDs, tables, and columns.
- **Strings:** Strings can be either literal strings or expressions with VARCHAR or CHAR data types.
- **Expressions:** Expressions are formed from several elements, like constants, SQL operators, column names, and subqueries.
- **Search Conditions:** Conditions are used to select a subset of the rows from a table or used to control statements like an IF statement to determine control of flow.
- **Special Values:** Special values should be used in expressions and as column defaults when building tables.
- **Variables:** Sybase IQ supports local variables, global variables, and connection-level variables.
- **Comments:** Comment is another SQL element which is used to attach explanatory text to SQL statements or blocks of statements. The database server does not execute any comment.
- **NULL Value:** Use NULL, which helps you to specify a value that is unknown, missing, or not applicable.

What is a database in SQL?

A database in SQL Server that is made up of a collection of tables that stores a detailed set of structured data. It is a table that contains a collection of rows, referred to as records or tuples, and columns that are also referred to as attributes.

Each column in the table is designed to store a specific type of information, for example, names, dates, dollar amounts, and numbers.

Data Manipulation Language:

Data Manipulation Language (DML) commands in SQL deals with manipulation of data records stored within the database tables. It does not deal with changes to database objects and its structure. The commonly known DML commands are INSERT, UPDATE and DELETE.

Command	Description
INSERT	Used to insert new data records or rows in the database table
UPDATE	Used to set the value of a field or column for a particular record to a new value
DELETE	Used to remove one or more rows from the database table

Commands of DML

Now let us try to understand each of the above mentioned DML commands in detail one by one.

1. INSERT

INSERT commands in SQL are used to insert data records or rows in a database table. In an INSERT statement, we specify both the column_names for which the entry has to be made along with the data value that has to be inserted.

The basic syntax for writing INSERT statements in SQL is as follows :

```
INSERT INTO table_name (column_name_1, column_name_2, column_name_3, ...)
VALUES (value1, value2, value3, ...)
```

By VALUES, we mean the value of the corresponding columns.

Here are a few examples to further illustrate the INSERT statement.

```
INSERT INTO public.customers( customer_id, sale_date, sale_amount, salesperson, store_state,
order_id) VALUES (1005,'2019-12-12',4200,'R K Rakesh','MH','1007');
```

Suppose if we have to insert values into all the fields of the database table, then we need not specify the column names, unlike the previous query. Follow the following query for further illustration.

```
INSERT INTO customers VALUES ('1006','2020-03-04',3200,'DL', '1008');
```

2. UPDATE

UPDATE command or statement is used to modify the value of an existing column in a database table.

The syntax for writing an UPDATE statement is as follows :

```
UPDATE table_name SET column_name_1 = value1, column_name_2 = value2, ...
WHERE condition;
```

```
UPDATE customers SET store_state = 'DL' WHERE store_state = 'NY';
```

In this example, we have modified the value of store_state for a record where store_state was 'NY' and set it to a new value 'DL'.

3. DELETE

DELETE statement in SQL is used to remove one or more rows from the database table. It does not delete the data records permanently. We can always perform a rollback operation to undo a DELETE command. With DELETE statements we can use the WHERE clause for filtering specific rows.

The syntax for writing an DELETE statement is as follows :

```
DELETE FROM table_name WHERE condition;
```

```
DELETE FROM customers WHERE store_state = 'MH' AND customer_id = '1001';
```

Differences between UNION EXCEPT and INTERSECT Operators:

1. The SET operators are mainly used to combine the result of more than 1 select statement and return a single result set to the user.
2. The set operators work on complete rows of the queries, so the results of the queries must have the same column name, same column order and the types of columns must be compatible.

There are the following 4 set operators in SQL Server:

1. **UNION**: Combine two or more result sets into a single set, without duplicates.
2. **UNION ALL**: Combine two or more result sets into a single set, including all duplicates.
3. **INTERSECT**: Takes the data from both result sets which are in common.
4. **EXCEPT**: Takes the data from the first result set, but not in the second result set (i.e. no matching to each other)

Rules on Set Operations:

1. The result sets of all queries must have the same number of columns.
2. In every result set the data type of each column must be compatible (well matched) to the data type of its corresponding column in other result sets.
3. In order to sort the result, an ORDER BY clause should be part of the last select statement. The column names or aliases must be found out by the first select statement.

Understand the differences between these operators with examples.

Use below SQL Script to create and populate the two tables that we are going to use in our examples.

```
mysql> create table colors_a(color_name varchar(20));
```

Query OK, 0 rows affected (0.17 sec)

```
mysql> create table colors_b(color_name varchar(20));
```

Query OK, 0 rows affected (0.11 sec)

```
mysql> insert into colors_a(color_name)values('red'),('green'),('orange'),('yellow'),('violet');
```

Query OK, 5 rows affected (0.09 sec)

Records: 5 Duplicates: 0 Warnings: 0

```
mysql> insert into colors_b(color_name)values('white'),('red'),('peach'),('orange');
```

Query OK, 4 rows affected (0.08 sec)

Records: 4 Duplicates: 0 Warnings: 0

```
mysql> select * from colors_a;
```

```
mysql> select * from colors_b;
```

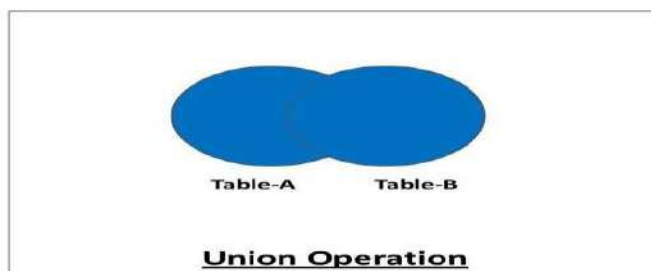
UNION

The Union is a binary set operator in DBMS. It is used to combine the result set of two select queries. Thus, It combines two result sets into one. In other words, the result set obtained after union operation is the collection of the result set of both the tables.

But two necessary conditions need to be fulfilled when we use the union command. These are:

1. Both SELECT statements should have an equal number of fields in the same order.
2. The data types of these fields should either be the same or compatible with each other.

The Union operation can be demonstrated as follows:



The syntax for the union operation is as follows:

```
SELECT (column_names) from table1 [WHERE condition] UNION SELECT  
(column_names) from table2 [WHERE condition];
```

The MySQL query for the union operation can be as follows:

```
mysql> select color_name from colors_a union select color_name from colors_b;
```

The Union operation gives us distinct values. If we want to allow the duplicates in our result set, we'll have to use the '**Union-All**' operation.

Union All operation is also similar to the union operation. The only difference is that it allows duplicate values in the result set.

The syntax for the union all operation is as follows:

```
SELECT (column_names) from table1 [WHERE condition] UNION ALL SELECT  
(column_names) from table2 [WHERE condition];
```

The MySQL query for the union all operation can be as follows:

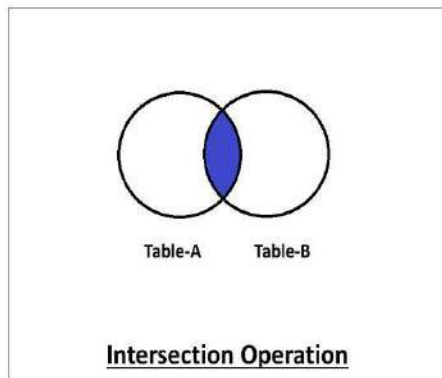
```
mysql> select color_name from colors_a union all select color_name from colors_b;
```

INTERSECT

Intersect is a binary set operator in DBMS. The intersection operation between two selections returns only the common data sets or rows between them. It should be noted that the intersection operation always returns the distinct rows. The duplicate rows will not be returned by the intersect operator.

Here also, the above conditions of the union and minus are followed, i.e., the number of fields in both the SELECT statements should be the same, with the same data type, and in the same order for the intersection.

The intersection operation can be demonstrated as follows:



The syntax for the intersection operation is as follows:

```
SELECT (column_names) from table1[WHERE condition] INTERSECT SELECT  
(column_names) from table2 [WHERE condition];
```

Note: It is to be noted that the intersect operator is not present in MySQL. But we can make use of 'IN' operator for performing an intersection operation in MySQL.

Here, we are using the 'IN' clause for demonstrating the examples.

The MySQL query for the intersection operation using the 'IN' operator can be as follows:

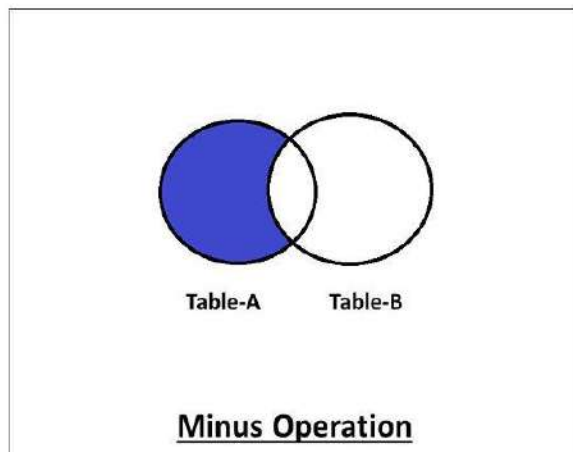
```
mysql> select color_name from colors_a where color_name in(select color_name from  
colors_b);
```

MINUS(EXCEPT)

Minus is a binary set operator in DBMS. The minus operation between two selections returns the rows that are present in the first selection but not in the second selection. The Minus operator returns only the distinct rows from the first table.

It is a must to follow the above conditions that we've seen in the union, i.e., the number of fields in both the SELECT statements should be the same, with the same data type, and in the same order for the minus operation.

The minus operation can be demonstrated as follows:



The syntax for the minus operation is as follows:

```
SELECT (column_names) from table1 [WHERE condition] MINUS/EXCEPT SELECT  
(column_names) from table2 [WHERE condition];
```

Note: It is to be noted that the minus operator is not present in MySQL. But we can make use of either 'NOT IN' operator or 'JOIN' for performing a minus operation in MySQL.

Here, we first see the 'NOT IN' operator for demonstrating the examples.

The MySQL query for the minus operation using the 'NOT IN' operator can be as follows:

```
mysql> select color_name from colors_a where color_name not in(select color_name from  
colors_b);
```

```
mysql> select color_name from colors_b where color_name not in(select color_name from  
colors_a);
```

Nested Queries:

A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause.

A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.

Important Rule:

- A subquery can be placed in a number of SQL clauses like WHERE clause, FROM clause, HAVING clause.
- You can use Subquery with SELECT, UPDATE, INSERT, DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.
- A subquery is a query within another query. The outer query is known as the main query, and the inner query is known as a subquery.
- Subqueries are on the right side of the comparison operator.
- A subquery is enclosed in parentheses.
- In the Subquery, ORDER BY command cannot be used. But GROUP BY command can be used to perform the same function as ORDER BY command.

1. Subqueries with the Select Statement

SQL subqueries are most frequently used with the Select statement.

Syntax

SELECT column_name FROM table_name WHERE column_name expression
operator (SELECT column_name from table_name WHERE ...);

Example

Consider the EMPLOYEE table have the following records:

ID	NAME	AGE	ADDRESS	SALARY
1	John	20	US	2000.00
2	Stephan	26	Dubai	1500.00
3	David	27	Bangkok	2000.00
4	Alina	29	UK	6500.00
5	Kathrin	34	Bangalore	8500.00
6	Harry	42	China	4500.00

7	Jackson	25	Mizoram	10000.00
---	---------	----	---------	----------

The subquery with a SELECT statement will be:

```
SELECT * FROM EMPLOYEE WHERE ID IN (SELECT ID FROM EMPLOYEE
WHERE SALARY > 4500);
```

This would produce the following result:

ID	NAME	AGE	ADDRESS	SALARY
4	Alina	29	UK	6500.00
5	Kathrin	34	Bangalore	8500.00
7	Jackson	25	Mizoram	10000.00

2. Subqueries with the INSERT Statement

- SQL subquery can also be used with the Insert statement. In the insert statement, data returned from the subquery is used to insert into another table.
- In the subquery, the selected data can be modified with any of the character, date functions.

Syntax:

```
INSERT INTO table_name (column1, column2, column3....)
```

```
SELECT * FROM table_name WHERE VALUE OPERATOR
```

Example

Consider a table EMPLOYEE_BKP with similar as EMPLOYEE.

Now use the following syntax to copy the complete EMPLOYEE table into the EMPLOYEE_BKP table.

```
INSERT INTO EMPLOYEE_BKP SELECT * FROM EMPLOYEE WHERE ID IN (SELECT ID
FROM EMPLOYEE);
```

3. Subqueries with the UPDATE Statement

The subquery of SQL can be used in conjunction with the Update statement. When a subquery is used with the Update statement, then either single or multiple columns in a table can be updated.

Syntax

```
UPDATE table SET column_name = new_value WHERE VALUE OPERATOR  
(SELECT COLUMN_NAME FROM TABLE_NAME WHERE condition);
```

Example

Let's assume we have an EMPLOYEE_BKP table available which is backup of EMPLOYEE table. The given example updates the SALARY by .25 times in the EMPLOYEE table for all employee whose AGE is greater than or equal to 29.

```
UPDATE EMPLOYEE SET SALARY = SALARY * 0.25 WHERE AGE IN (SELECT AGE FR  
OM EMPLOYEE_BKP WHERE AGE >= 29);
```

This would impact three rows, and finally, the EMPLOYEE table would have the following records.

ID	NAME	AGE	ADDRESS	SALARY
1	John	20	US	2000.00
2	Stephan	26	Dubai	1500.00
3	David	27	Bangkok	2000.00
4	Alina	29	UK	1625.00
5	Kathrin	34	Bangalore	2125.00

6	Harry	42	China	1125.00
7	Jackson	25	Mizoram	10000.00

4. Subqueries with the DELETE Statement

The subquery of SQL can be used in conjunction with the Delete statement just like any other statements mentioned above.

Syntax

DELETE FROM TABLE_NAME WHERE VALUE OPERATOR (SELECT COLUMN_NAME FROM TABLE_NAME WHERE condition);

Example

Let's assume we have an EMPLOYEE_BKP table available which is backup of EMPLOYEE table. The given example deletes the records from the EMPLOYEE table for all EMPLOYEE whose AGE is greater than or equal to 29.

DELETE FROM EMPLOYEE WHERE AGE IN (SELECT AGE FROM EMPLOYEE_BKP WHERE AGE >= 29);

This would impact three rows, and finally, the EMPLOYEE table would have the following records.

ID	NAME	AGE	ADDRESS	SALARY
1	John	20	US	2000.00
2	Stephan	26	Dubai	1500.00
3	David	27	Bangkok	2000.00
7	Jackson	25	Mizoram	10000.00

Aggregate Functions in DBMS:

Aggregate functions in DBMS take multiple rows from the table and return a value according to the query.

All the aggregate functions are used in Select statement.

Syntax –

```
SELECT <FUNCTION NAME> (<PARAMETER>) FROM <TABLE NAME>
```

```
mysql> create table employee(id int,name varchar(20),age int,address varchar(20),salary decimal(10,2));
```

Query OK, 0 rows affected (0.25 sec)

```
mysql> insert into employee(id,name,age,address,salary)values(1,'john',20,'us',2000.00),(2,'stephan',26,'dubai',1500.00),(3,'david',27,'bangkok',2000.00),(4,'alina',29,'uk',6500.00),(5,'kathrin',34,'bangalore',8500.00),(6,'harry',42,'china',4500.00),(7,'jackson',25,'mizoram',10000.00);
```

Query OK, 7 rows affected (0.05 sec)

Records: 7 Duplicates: 0 Warnings: 0

```
mysql> select * from employee;
```

```
+-----+-----+-----+-----+-----+
| ID | NAME | AGE | ADDRESS | SALARY |
+-----+-----+-----+-----+-----+
| 1 | john | 20 | us | 2000.00 |
| 2 | stephan | 26 | dubai | 1500.00 |
| 3 | david | 27 | bangkok | 2000.00 |
| 4 | alina | 29 | uk | 6500.00 |
| 5 | kathrin | 34 | bangalore | 8500.00 |
| 6 | harry | 42 | china | 4500.00 |
| 7 | jackson | 25 | mizoram | 10000.00 |
+-----+-----+-----+-----+-----+
```

7 rows in set (0.00 sec)

AVG Function:

This function returns the average value of the numeric column that is supplied as a parameter.

Example: Write a query to select average salary from employee table.

```
Select AVG(salary) from employee;
```

```
mysql> Select AVG(salary) from employee;
```

```
+-----+
```

```
| AVG(salary) |
```

```
+-----+
```

```
| 5000.000000 |
```

```
+-----+
```

```
1 row in set (0.00 sec)
```

COUNT Function:

The count function returns the number of rows in the result. It does not count the null values.

Example: Write a query to return number of rows where salary > 20000.

```
Select COUNT(*) from employee where salary > 2000;
```

Types –

- COUNT(*): Counts all the number of rows of the table including null.
- COUNT(COLUMN_NAME): count number of non-null values in column.
- COUNT(DISTINCT COLUMN_NAME): count number of distinct values in a column.

```
mysql> Select COUNT(*) from employee where salary > 2000;
```

```
+-----+
```

```
| COUNT(*) |
```

```
+-----+
```

```
|    4 |
```

```
+-----+
```

```
1 row in set (0.00 sec)
```

MAX Function:

The MAX function is used to find maximum value in the column that is supplied as a parameter. It can be used on any type of data.

Example – Write a query to find the maximum salary in employee table.

```
Select MAX(salary) from employee;
```

```
mysql> Select MAX(salary) from employee;
```

```
+-----+
```

```
| MAX(salary) |
```

```
+-----+
```

```
| 10000.00 |
```

```
+-----+
```

```
1 row in set (0.00 sec)
```

MIN Function:

The MIN function is used to find minimum value in the column that is supplied as a parameter. It can be used on any type of data.

Example – Write a query to find the minimum salary in employee table.

```
Select MIN(salary) from employee;
```

```
mysql> Select MIN(salary) from employee;
```

```
+-----+
```

```
| MIN(salary) |
```

```
+-----+
```

```
| 1500.00 |
```

```
+-----+
```

```
1 row in set (0.00 sec)
```

SUM Function:

This function sums up the values in the column supplied as a parameter.

Example: Write a query to get the total salary of employees.

```
Select SUM(salary) from employee;
```

```
mysql> Select SUM(salary) from employee;
```

```
+-----+
```

```
| SUM(salary) |
```

```
+-----+
```

```
| 35000.00 |
```

```
+-----+
```

```
1 row in set (0.00 sec)
```

SQL - NULL Values

The SQL **NULL** is the term used to represent a missing value. A NULL value in a table is a value in a field that appears to be blank.

A field with a NULL value is a field with no value. It is very important to understand that a NULL value is different than a zero value or a field that contains spaces.

Syntax

The basic syntax of **NULL** while creating a table.

```
mysql> create table customers( id int not null, name varchar (20) not null, age int not null, address char (25) , salary decimal (18, 2), primary key (id));
```

```
Query OK, 0 rows affected (0.22 sec)
```

```
mysql> insert into customers (id,name,age,address,salary) values  
(1,'ramesh',32,'ahmedabad',2000.00),(2,'khilan',25,'delhi',1500.00),(3,'kaushik',23,'kota',2000.00),  
(4,'chaitali',25,'mumbai',6500.00),(5,'hardik',27,'bhopal',8500.00);
```

```
Query OK, 5 rows affected (0.06 sec)
```

```
Records: 5 Duplicates: 0 Warnings: 0
```



```
mysql> insert into customers (id,name,age,address) values  
(6,'komal',22,'mp'),(7,'muffy',24,'indore');
```

Query OK, 2 rows affected (0.12 sec)

Records: 2 Duplicates: 0 Warnings: 0

```
mysql> select * from customers;
```

```
+----+-----+----+-----+-----+  
| id | name  | age | address | salary |  
+----+-----+----+-----+-----+  
| 1 | ramesh | 32 | ahmedabad | 2000.00 |  
| 2 | khilan | 25 | delhi    | 1500.00 |  
| 3 | kaushik | 23 | kota     | 2000.00 |  
| 4 | chaitali | 25 | mumbai   | 6500.00 |  
| 5 | hardik | 27 | bhopal   | 8500.00 |  
| 6 | komal  | 22 | mp       | NULL    |  
| 7 | muffy  | 24 | indore   | NULL    |  
+----+-----+----+-----+-----+
```

7 rows in set (0.00 sec)

Here, **NOT NULL** signifies that column should always accept an explicit value of the given data type. There are two columns where we did not use NOT NULL, which means these columns could be NULL.

A field with a NULL value is the one that has been left blank during the record creation.

Example

The NULL value can cause problems when selecting data. However, because when comparing an unknown value to any other value, the result is always unknown and not included in the results. You must use the **IS NULL** or **IS NOT NULL** operators to check for a NULL value.

Consider the following customers table having the records as shown below.

```
mysql> select * from customers;
```

```
+----+-----+----+-----+-----+
| id | name   | age | address | salary |
+----+-----+----+-----+-----+
| 1 | ramesh | 32 | ahmedabad | 2000.00 |
| 2 | khilan | 25 | delhi     | 1500.00 |
| 3 | kaushik | 23 | kota      | 2000.00 |
| 4 | chaitali | 25 | mumbai    | 6500.00 |
| 5 | hardik | 27 | bhopal    | 8500.00 |
| 6 | komal | 22 | mp        | NULL |
| 7 | muffy | 24 | indore    | NULL |
+----+-----+----+-----+-----+
```

```
7 rows in set (0.00 sec)
```

Now, following is the usage of the **IS NOT NULL** operator.

```
mysql> select id, name, age, address, salary from customers where salary is not null;
```

```
+----+-----+----+-----+-----+
| id | name   | age | address | salary |
+----+-----+----+-----+-----+
| 1 | ramesh | 32 | ahmedabad | 2000.00 |
| 2 | khilan | 25 | delhi     | 1500.00 |
| 3 | kaushik | 23 | kota      | 2000.00 |
| 4 | chaitali | 25 | mumbai    | 6500.00 |
| 5 | hardik | 27 | bhopal    | 8500.00 |
+----+-----+----+-----+-----+
```

5 rows in set (0.00 sec)

Now, following is the usage of the **IS NULL** operator.

```
mysql> select id, name, age, address, salary from customers where salary is null;
```

```
+----+-----+----+-----+-----+
| id | name  | age | address | salary |
+----+-----+----+-----+-----+
| 6  | komal | 22  | mp      | NULL   |
| 7  | muffy | 24  | indore  | NULL   |
+----+-----+----+-----+-----+
```

2 rows in set (0.00 sec)

Complex integrity Constraints in SQL:

Constraints are the rules enforced on the data columns of a table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database.

Constraints could be either on a column level or a table level. The column level constraints are applied only to one column, whereas the table level constraints are applied to the whole table.

Following are some of the most commonly used constraints available in SQL.

- NOT NULL Constraint – Ensures that a column cannot have NULL value.

For example, the below query creates a table Student with the fields ID and NAME as NOT NULL. That is, we are bound to specify values for these two fields every time we wish to insert a new row.

```
CREATE TABLE Student (ID int(6) NOT NULL, NAME varchar(10) NOT NULL, ADDRESS varchar(20));
```

- DEFAULT Constraint – Provides a default value for a column when none is specified.

For example, the below query will create a table named Student and specify the default value for the field AGE as 18.

```
CREATE TABLE Student (ID int(6) NOT NULL,NAME varchar(10) NOT NULL,  
AGE int DEFAULT 18);
```

- UNIQUE Constraint – Ensures that all values in a column are different.

For example, the below query creates a table Student where the field ID is specified as UNIQUE. i.e, no two students can have the same ID. Unique constraint in detail.

```
CREATE TABLE Student(ID int(6) NOT NULL UNIQUE,NAME varchar(10),  
ADDRESS varchar(20));
```

- CHECK Constraint – The CHECK constraint ensures that all the values in a column satisfies certain conditions.

For example, the below query creates a table Student and specifies the condition for the field AGE as (AGE >= 18). That is, the user will not be allowed to enter any record in the table with AGE < 18.

```
CREATE TABLE Student (ID int(6) NOT NULL,NAME varchar(10) NOT NULL,  
AGE int NOT NULL CHECK (AGE >= 18));
```

Trigger:

Triggers are the SQL statements that are **automatically executed** when there is any change in the database. The triggers are executed **in response to certain events** (INSERT, UPDATE or DELETE) in a particular table. These triggers help in maintaining the integrity of the data by changing the data of the database in a systematic fashion.

Syntax

```
create trigger Trigger_name (before | after) [insert | update | delete] on [table_name] [for each  
row] [trigger_body];
```

Join Operations:

In DBMS, a join statement is mainly used to combine two tables based on a specified common field between them. If we talk in terms of Relational algebra, it is the cartesian product of two tables followed by the selection operation. Thus, we can execute the product and selection process on two tables using a single join statement. We can use either 'on' or 'using' clause in MySQL to apply predicates to the join queries.

A Join can be broadly divided into two types:

- 1. Inner Join**
- 2. Outer Join**

For all the examples, we will consider the below-mentioned employee and department table.

```
mysql> create table employee (empId int,empName varchar(20),deptId int);
```

```
Query OK, 0 rows affected (0.16 sec)
```

```
mysql> insert into employee (empId,empName,deptId) values  
(1,'Harry',2),(2,'Tom',3),(3,'Joy',5),(4,'Roy',8);
```

```
Query OK, 4 rows affected (0.06 sec)
```

```
Records: 4 Duplicates: 0 Warnings: 0
```

```
mysql> create table department(deptId int, deptName varchar(20));
```

```
Query OK, 0 rows affected (0.14 sec)
```

```
mysql> insert into department(deptId,deptName) values (1,'CSE'),(2,'Mech'),(3,'IT');
```

```
Query OK, 3 rows affected (0.07 sec)
```

```
Records: 3 Duplicates: 0 Warnings: 0
```

```
mysql> select * from employee;
```

```
+-----+-----+-----+  
| empId | empName | deptId |  
+-----+-----+-----+  
| 1 | Harry | 2 |
```

	2	Tom		3	
--	---	-----	--	---	--

	3	Joy		5	
--	---	-----	--	---	--

	4	Roy		8	
--	---	-----	--	---	--

+-----+-----+-----+

4 rows in set (0.05 sec)

mysql> select * from department;

+-----+-----+

	deptId		deptName	
--	--------	--	----------	--

+-----+-----+

	1	CSE	
--	---	-----	--

	2	Mech	
--	---	------	--

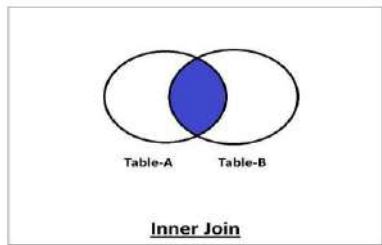
	3	IT	
--	---	----	--

+-----+-----+

3 rows in set (0.00 sec)

Inner Join

Inner Join is a join that can be used to return all the values that have matching values in both the tables. Inner Join can be depicted using the below diagram.



The inner join can be further divided into the following types:

1. **Equi Join**
2. **Natural Join**

1. Equi Join

Equi Join is an inner join that uses the equivalence condition for fetching the values of two tables.

Query:

```
mysql> Select employee.empId, employee.empName, department.deptName from employee  
inner join department on employee.deptId = department.deptId;
```

2. Natural Join

Natural Join is an inner join that returns the values of the two tables on the basis of a common attribute that has the same name and domain. It does not use any comparison operator. It also removes the duplicate attribute from the results.

Query:

```
mysql> select * from employee natural join department;
```

The above query will return the values of tables removing the duplicates.
If we want to specify the attribute names, the query will be as follows:

Query:

```
mysql> Select employee.empId, employee.empName, department.deptId, department.deptName  
from employee natural join department;
```

Outer Join

Outer Join is a join that can be used to return the records in both the tables whether it has matching records in both the tables or not.

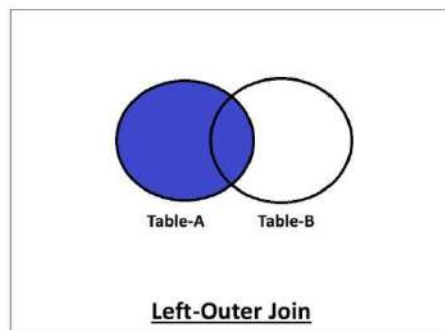
The outer join can be further divided into three types:

1. **Left-Outer Join**
2. **Right-Outer Join**
3. **Full-Outer Join**

1. Left-Outer Join:

The Left-Outer Join is an outer join that returns all the values of the left table, and the values of the right table that has matching values in the left table.

If there is no matching result in the right table, it will return null values in that field. The Left-Outer Join can be depicted using the below diagram.



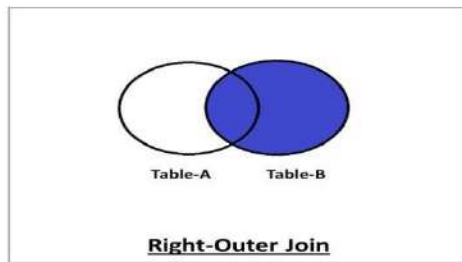
Query:

```
mysql> select employee.empId, employee.empName, department.deptName from  
employee left outer join department on employee.deptId = department.deptId;
```

2. Right-Outer Join:

The Right-Outer Join is an outer join that returns all the values of the right table, and the values of the left table that has matching values in the right table.

The Right-Outer Join can be depicted using the below diagram.



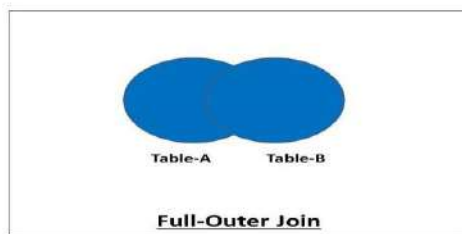
Query:

```
mysql> select employee.empId, employee.empName, department.deptName from employee
right outer join department on employee.deptId = department.deptId;
```

3. Full-Outer Join:

The Full-Outer join contains all the values of both the tables whether they have matching values in them or not.

The Full-Outer Join can be depicted using the below diagram.



Query:

```
mysql> select * from employee full join department;
```

Conditional Join (\bowtie_c): Conditional Join is used when you want to join two or more relation based on some conditions.

Division Operator (\div): Division operator $A \div B$ can be applied if and only if:

- Attributes of B is proper subset of Attributes of A.
- The relation returned by division operator will have attributes = (All attributes of A – All Attributes of B)
- The relation returned by division operator will return those tuples from relation A which are associated to every B's tuple.

Schema Refinement:

Redundancy:

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

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

The Problem of redundancy in Database

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

1. Insertion Anomaly –

If a student detail has to be inserted whose course is not being decided yet then insertion will not be possible till the time course is decided for student.

Student_ID	Name	Contact	College	Course	Rank
100	Himanshu	7300934851	GEU		1

This problem happens when the insertion of a data record is not possible without adding some additional unrelated data to the record.

2. Deletion Anomaly –

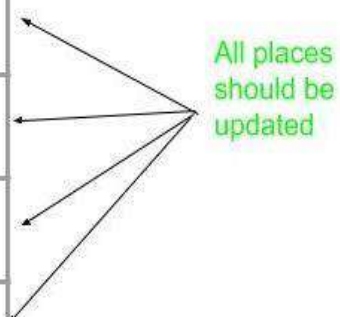
If the details of students in this table is deleted then the details of college will also get deleted which should not occur by common sense.

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

3. Updation Anomaly –

Suppose if the rank of the college changes then changes will have to be all over the database which will be time-consuming and computationally costly.

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



All places should be updated

If updation do not occur at all places then database will be in inconsistent state.

What is Functional Dependency?

Functional Dependency (FD) is a constraint that determines the relation of one attribute to another attribute in a Database Management System (DBMS). Functional Dependency helps to maintain the quality of data in the database. It plays a vital role to find the difference between good and bad database design.

A functional dependency is denoted by an arrow " \rightarrow ". The functional dependency of X on Y is represented by $X \rightarrow Y$. Let's understand Functional Dependency in DBMS with example.

Example:

Employee number	Employee Name	Salary	City
1	Dana	50000	San Francisco
2	Francis	38000	London

3	Andrew	25000	Tokyo
---	--------	-------	-------

In this example, if we know the value of Employee number, we can obtain Employee Name, city, salary, etc. By this, we can say that the city, Employee Name, and salary are functionally depended on Employee number.

Key terms

Here, are some key terms for Functional Dependency in Database:

Key Terms	Description
Axiom	Axioms is a set of inference rules used to infer all the functional dependencies on a relational database.
Decomposition	It is a rule that suggests if you have a table that appears to contain two entities which are determined by the same primary key then you should consider breaking them up into two different tables.
Dependent	It is displayed on the right side of the functional dependency diagram.
Determinant	It is displayed on the left side of the functional dependency Diagram.
Union	It suggests that if two tables are separate, and the PK is the same, you should consider putting them. together

Rules of Functional Dependencies

Below are the Three most important rules for Functional Dependency in Database:

- Reflexive rule –. If X is a set of attributes and Y is_subset_of X, then X holds a value of Y.
- Augmentation rule: When $x \rightarrow y$ holds, and c is attribute set, then $ac \rightarrow bc$ also holds. That is adding attributes which do not change the basic dependencies.
- Transitivity rule: This rule is very much similar to the transitive rule in algebra if $x \rightarrow y$ holds and $y \rightarrow z$ holds, then $x \rightarrow z$ also holds. $X \rightarrow y$ is called as functionally that determines y.

Types of Functional Dependencies in DBMS

There are mainly four types of Functional Dependency in DBMS. Following are the types of Functional Dependencies in DBMS:

- **Multivalued Dependency**
- **Trivial Functional Dependency**
- **Non-Trivial Functional Dependency**
- **Transitive Dependency**

Multivalued Dependency in DBMS:

Multivalued dependency occurs in the situation where there are multiple independent multivalued attributes in a single table. A multivalued dependency is a complete constraint between two sets of attributes in a relation. It requires that certain tuples be present in a relation. Consider the following Multivalued Dependency Example to understand.

Example:

Car_model	Maf_year	Color
H001	2017	Metallic
H001	2017	Green
H005	2018	Metallic
H005	2018	Blue
H010	2015	Metallic
H033	2012	Gray

In this example, maf_year and color are independent of each other but dependent on car_model. In this example, these two columns are said to be multivalued dependent on car_model.

This dependence can be represented like this:

car_model \twoheadrightarrow maf_year

car_model \twoheadrightarrow colour

Trivial Functional Dependency in DBMS:

The Trivial dependency is a set of attributes which are called a trivial if the set of attributes are included in that attribute.

So, $X \rightarrow Y$ is a trivial functional dependency if Y is a subset of X . Let's understand with a Trivial Functional Dependency Example.

For example:

Emp_id	Emp_name
AS555	Harry
AS811	George
AS999	Kevin

Consider this table with two columns Emp_id and Emp_name.

$\{Emp_id, Emp_name\} \rightarrow Emp_id$ is a trivial functional dependency as Emp_id is a subset of $\{Emp_id, Emp_name\}$.

Non Trivial Functional Dependency in DBMS:

Functional dependency which is also known as a nontrivial dependency occurs when $A \rightarrow B$ holds true where B is not a subset of A . In a relationship, if attribute B is not a subset of attribute A , then it is considered as a non-trivial dependency.

Company	CEO	Age
Microsoft	Satya Nadella	51
Google	Sundar Pichai	46

Apple	Tim Cook	57
-------	----------	----

Example:

{Company} → {CEO} (if we know the Company, we know the CEO name)

But CEO is not a subset of Company, and hence it's non-trivial functional dependency.

Transitive Dependency in DBMS:

A Transitive Dependency is a type of functional dependency which happens when it is indirectly formed by two functional dependencies. Let's understand with the following Transitive Dependency Example.

Example:

Company	CEO	Age
Microsoft	Satya Nadella	51
Google	Sundar Pichai	46
Alibaba	Jack Ma	54

{Company} → {CEO} (if we know the company, we know its CEO's name)

{CEO} → {Age} If we know the CEO, we know the Age

Therefore according to the rule of transitive dependency:

{Company} → {Age} should hold, that makes sense because if we know the company name, we can know his age.

Note: You need to remember that transitive dependency can only occur in a relation of three or more attributes.

Decomposition

Decomposition in DBMS removes redundancy, anomalies and inconsistencies from a database by dividing the table into multiple tables.

The following are the types –

Lossless Join Decomposition:

Decomposition is lossless if it is feasible to reconstruct relation R from decomposed tables using Joins. This is the preferred choice. The information will not lose from the relation when decomposed. The join would result in the same original relation.

Let us see an example –

<EmpInfo>

Emp_ID	Emp_Name	Emp_Age	Emp_Location	Dept_ID	Dept_Name
E001	Jacob	29	Alabama	Dpt1	Operations
E002	Henry	32	Alabama	Dpt2	HR
E003	Tom	22	Texas	Dpt3	Finance

Decompose the above table into two tables:

<EmpDetails>

Emp_ID	Emp_Name	Emp_Age	Emp_Location
E001	Jacob	29	Alabama
E002	Henry	32	Alabama
E003	Tom	22	Texas

<DeptDetails>

Dept_ID	Emp_ID	Dept_Name
Dpt1	E001	Operations
Dpt2	E002	HR
Dpt3	E003	Finance

Now, Natural Join is applied on the above two tables –

The result will be –

Emp_ID	Emp_Name	Emp_Age	Emp_Location	Dept_ID	Dept_Name
E001	Jacob	29	Alabama	Dpt1	Operations
E002	Henry	32	Alabama	Dpt2	HR
E003	Tom	22	Texas	Dpt3	Finance

Therefore, the above relation had lossless decomposition i.e. no loss of information.

Lossy Join Decomposition:

As the name suggests, when a relation is decomposed into two or more relational schemas, the loss of information is unavoidable when the original relation is retrieved.

Let us see an example –

<EmpInfo>

Emp_ID	Emp_Name	Emp_Age	Emp_Location	Dept_ID	Dept_Name
E001	Jacob	29	Alabama	Dpt1	Operations
E002	Henry	32	Alabama	Dpt2	HR
E003	Tom	22	Texas	Dpt3	Finance

Decompose the above table into two tables –

<EmpDetails>

Emp_ID	Emp_Name	Emp_Age	Emp_Location

E001	Jacob	29	Alabama
E002	Henry	32	Alabama
E003	Tom	22	Texas

<DeptDetails>

Dept_ID	Dept_Name
Dpt1	Operations
Dpt2	HR
Dpt3	Finance

Now, you won't be able to join the above tables, since **Emp_ID** isn't part of the **DeptDetails** relation.

Therefore, the above relation has lossy decomposition.

What is Normalization?

Normalization is a method of organizing the data in the database which helps you to avoid data redundancy, insertion, update & deletion anomaly. It is a process of analyzing the relation schemas based on their different functional dependencies and primary key.

Normalization is inherent to relational database theory. It may have the effect of duplicating the same data within the database which may result in the creation of additional tables.

Advantages of Functional Dependency

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

Normalization is a process of organizing the data in database to avoid data redundancy, insertion anomaly, update anomaly & deletion anomaly. Let's discuss about anomalies first then we will discuss normal forms with examples.

Anomalies in DBMS

There are three types of anomalies that occur when the database is not normalized. These are – Insertion, update and deletion anomaly. Let's take an example to understand this.

Example: Suppose a manufacturing company stores the employee details in a table named employee that has four attributes: emp_id for storing employee's id, emp_name for storing employee's name, emp_address for storing employee's address and emp_dept for storing the department details in which the employee works. At some point of time the table looks like this:

emp_id	emp_name	emp_address	emp_dept
101	Rick	Delhi	D001
101	Rick	Delhi	D002
123	Maggie	Agra	D890
166	Glenn	Chennai	D900
166	Glenn	Chennai	D004

- The above table is not normalized. We will see the problems that we face when a table is not normalized.

Update anomaly: In the above table we have two rows for employee Rick as he belongs to two departments of the company. If we want to update the address of Rick then we have to update the same in two rows or the data will become inconsistent. If somehow, the correct address gets

updated in one department but not in other then as per the database, Rick would be having two different addresses, which is not correct and would lead to inconsistent data.

Insert anomaly: Suppose a new employee joins the company, who is under training and currently not assigned to any department then we would not be able to insert the data into the table if emp_dept field doesn't allow nulls.

Delete anomaly: Suppose, if at a point of time the company closes the department D890 then deleting the rows that are having emp_dept as D890 would also delete the information of employee Maggie since she is assigned only to this department.

To overcome these anomalies we need to normalize the data.

Normalization

Here are the most commonly used normal forms:

- First normal form(1NF)
- Second normal form(2NF)
- Third normal form(3NF)
- Boyce & Codd normal form (BCNF)

First normal form (1NF)

As per the rule of first normal form, an attribute (column) of a table cannot hold multiple values. It should hold only atomic values.

Example: Suppose a company wants to store the names and contact details of its employees. It creates a table that looks like this:

emp_id	emp_name	emp_address	emp_mobile
101	Herschel	New Delhi	8912312390
102	Jon	Kanpur	8812121212 9900012222

103	Ron	Chennai	7778881212
104	Lester	Bangalore	9990000123 8123450987

Two employees (Jon & Lester) are having two mobile numbers so the company stored them in the same field as you can see in the table above.

This table is **not in 1NF** as the rule says “each attribute of a table must have atomic (single) values”, the emp_mobile values for employees Jon & Lester violates that rule.

To make the table complies with 1NF we should have the data like this:

emp_id	emp_name	emp_address	emp_mobile
101	Herschel	New Delhi	8912312390
102	Jon	Kanpur	8812121212
102	Jon	Kanpur	9900012222
103	Ron	Chennai	7778881212
104	Lester	Bangalore	9990000123
104	Lester	Bangalore	8123450987

Second normal form (2NF)

A table is said to be in 2NF if both the following conditions hold:

- Table is in 1NF (First normal form)
- No non-prime attribute is dependent on the proper subset of any candidate key of table.

An attribute that is not part of any candidate key is known as non-prime attribute.

Example: Suppose a school wants to store the data of teachers and the subjects they teach. They create a table that looks like this: Since a teacher can teach more than one subjects, the table can have multiple rows for a same teacher.

teacher_id	Subject	teacher_age
111	Maths	38
111	Physics	38
222	Biology	38
333	Physics	40
333	Chemistry	40

Candidate Keys: {teacher_id, subject}

Non prime attribute: teacher_age

The table is in 1 NF because each attribute has atomic values. However, it is not in 2NF because non prime attribute teacher_age is dependent on teacher_id alone which is a proper subset of candidate key. This violates the rule for 2NF as the rule says “**no** non-prime attribute is dependent on the proper subset of any candidate key of the table”.

To make the table complies with 2NF we can break it in two tables like this:
teacher_details table:

teacher_id	teacher_age
111	38
222	38
333	40

teacher_subject table:

teacher_id	subject
111	Maths
111	Physics
222	Biology
333	Physics

333	Chemistry
-----	-----------

Now the tables comply with Second normal form (2NF).

Third Normal form (3NF)

A table design is said to be in 3NF if both the following conditions hold:

- Table must be in 2NF
- Transitive functional dependency of non-prime attribute on any super key should be removed.

An attribute that is not part of any candidate key is known as non-prime attribute.

In other words 3NF can be explained like this: A table is in 3NF if it is in 2NF and for each functional dependency $X \rightarrow Y$ at least one of the following conditions hold:

- X is a super key of table
- Y is a prime attribute of table

An attribute that is a part of one of the candidate keys is known as prime attribute.

Example: Suppose a company wants to store the complete address of each employee, they create a table named employee_details that looks like this:

emp_id	emp_name	emp_zip	emp_state	emp_city	emp_district
1001	John	282005	UP	Agra	Dayal Bagh
1002	Ajeet	222008	TN	Chennai	M-City
1006	Lora	282007	TN	Chennai	Urrapakkam

1101	Lilly	292008	UK	Pauri	Bhagwan
1201	Steve	222999	MP	Gwalior	Ratan

Super keys: {emp_id}, {emp_id, emp_name}, {emp_id, emp_name, emp_zip}...so on

Candidate Keys: {emp_id}

Non-prime attributes: all attributes except emp_id are non-prime as they are not part of any candidate keys.

Here, emp_state, emp_city & emp_district dependent on emp_zip. And, emp_zip is dependent on emp_id that makes non-prime attributes (emp_state, emp_city & emp_district) transitively dependent on super key (emp_id). This violates the rule of 3NF.

To make this table complies with 3NF we have to break the table into two tables to remove the transitive dependency:

employee table:

emp_id	emp_name	emp_zip
1001	John	282005
1002	Ajeet	222008
1006	Lora	282007
1101	Lilly	292008

1201	Steve	222999
------	-------	--------

employee_zip table:

emp_zip	emp_state	emp_city	emp_district
282005	UP	Agra	Dayal Bagh
222008	TN	Chennai	M-City
282007	TN	Chennai	Urrapakkam
292008	UK	Pauri	Bhagwan
222999	MP	Gwalior	Ratan

Boyce Codd normal form (BCNF)

It is an advance version of 3NF that's why it is also referred as 3.5NF. BCNF is stricter than 3NF. A table complies with BCNF if it is in 3NF and for every functional dependency $X \rightarrow Y$, X should be the super key of the table.

Example: Suppose there is a company wherein employees work in **more than one department**. They store the data like this:

emp_id	emp_nationality	emp_dept	dept_type	dept_no_of_emp
1001	Austrian	Production and planning	D001	200
1001	Austrian	Stores	D001	250
1002	American	design and technical support	D134	100
1002	American	Purchasing department	D134	600

Functional dependencies in the table above:

emp_id -> emp_nationality

emp_dept -> {dept_type, dept_no_of_emp}

Candidate key: {emp_id, emp_dept}

The table is not in BCNF as neither emp_id nor emp_dept alone are keys.

To make the table comply with BCNF we can break the table in three tables like this:

emp_nationality table:

emp_id	emp_nationality
--------	-----------------

1001	Austrian
1002	American

emp_dept table:

emp_dept	dept_type	dept_no_of_emp
Production and planning	D001	200
Stores	D001	250
design and technical support	D134	100
Purchasing department	D134	600

emp_dept_mapping table:

emp_id	emp_dept
1001	Production and planning

1001	Stores
1002	design and technical support
1002	Purchasing department

Functional dependencies:

emp_id → emp_nationality

emp_dept → {dept_type, dept_no_of_emp}

Candidate keys:

For first table: emp_id

For second table: emp_dept

For third table: {emp_id, emp_dept}

This is now in BCNF as in both the functional dependencies left side part is a key.

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 $A \twoheadrightarrow B$, if for a single value of A, multiple values of B exists, then the relation will be a multi-valued dependency.

EXAMPLE

STUDENT

STU_ID	COURSE	HOBBY
21	Computer	Dancing
21	Math	Singing
34	Chemistry	Dancing
74	Biology	Cricket
59	Physics	Hockey

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:

STUDENT_COURSE

STU_ID	COURSE
21	Computer
21	Math
34	Chemistry
74	Biology
59	Physics

STUDENT_HOBBY

STU_ID	HOBBY
21	Dancing
21	Singing
34	Dancing
74	Cricket
59	Hockey

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

SUBJECT	LECTURER	SEMESTER
Computer	Anshika	Semester 1
Computer	John	Semester 1
Math	John	Semester 1
Math	Akash	Semester 2
Chemistry	Praveen	Semester 1

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:

P1

SEMESTER	SUBJECT
Semester 1	Computer
Semester 1	Math
Semester 1	Chemistry
Semester 2	Math

P2

SUBJECT	LECTURER
Computer	Anshika
Computer	John
Math	John
Math	Akash
Chemistry	Praveen

P3

SEMSTER	LECTURER
Semester 1	Anshika
Semester 1	John
Semester 1	John
Semester 2	Akash
Semester 1	Praveen