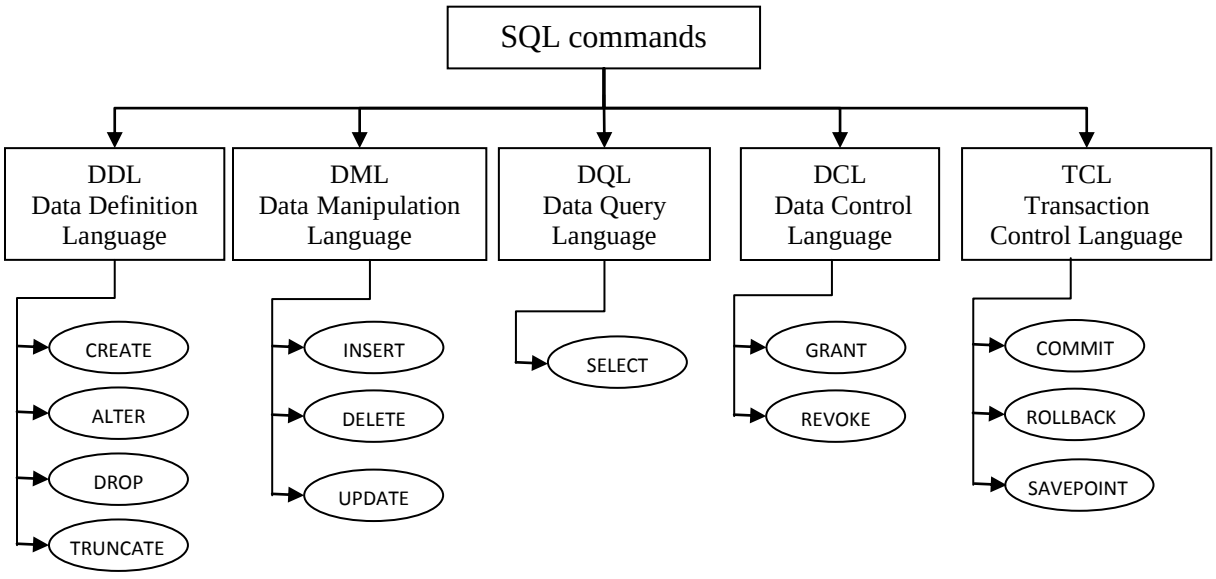
UNIT – III

SQL: QUERIES, CONSTRAINTS, TRIGGERS: form of basic SQL query, UNION, INTERSECT, and EXCEPT, Nested Queries, aggregation operators, NULL values, complex integrity constraints in SQL, triggers and active databases. **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**.**

1. SQL COMMANDS

Structured Query Language (SQL) is the database language used to create a database and to perform operations on the existing database. SQL commands are instructions used to communicate with the database to perform specific tasks and queries with data. These SQL commands are categorized into five categories as:

- **i.** DDL: Data Definition Language
- **ii.** DML: Data Manipulation Language
- **iii.** DQL: Data Query Language
- **iv.** DCL : Data Control Language
- **v.** TCL : Transaction Control Language.



i. **DDL(Data Definition Language) :** DDL or Data Definition Language consists of the SQL commands that can be used to define the database schema. It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database. The DQL commands are:

- **CREATE**: It is used to create the database or its objects (like table, index, function, views, store procedure and triggers).
- **DROP**: It is used to delete objects from the database.
- **ALTER**: It is used to alter the structure of the database.
- **TRUNCATE**: It is used to remove all records from a table, including all spaces allocated for the records are removed.

ii. **DQL (Data Query Language):** DML statements are used for performing queries on the data within schema objects. The purpose of DQL Command is to get data from some schema relation based on the query passed to it. The DQL commands are:

- **SELECT** – is used to retrieve data from the database.

iii. **DML (Data Manipulation Language):** The SQL commands that deals with the manipulation of data present in the database belong to DML or Data Manipulation Language and this includes most of the SQL statements. The DML commands are:

- **INSERT** – is used to insert data into a table.
- **UPDATE** – is used to update existing data within a table.
- **DELETE** – is used to delete records from a database table.

iv. **DCL (Data Control Language):** DCL includes commands which mainly deal with the rights, permissions and other controls of the database system. The DCL commands are:

- **GRANT**-gives user's access privileges to database.
- **REVOKE**-withdraw user's access privileges given by using the GRANT command.

v. **TCL (transaction Control Language):** TCL commands deals with the transaction within the database. The TCL commands are:

- **COMMIT**– commits a Transaction.
- **ROLLBACK**– rollbacks a transaction in case of any error occurs.
- **SAVEPOINT**–sets a save point within a transaction.

2. DDL COMMANDS

DDL or Data Definition Language consists of the SQL commands that can be used to define the database schema. It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database. The DQL commands are:

i. **CREATE**: It is used to create the database or its objects like table, index, function, views, store procedure and triggers.

a) **The 'CREATE DATABASE' Statement:** This statement is used to create a database.

Syntax: `CREATE DATABASE Database_Name;`

Example: `CREATE DATABASE Employee;`

It creates `Employee` database.

b) **The 'CREATE TABLE' Statement:** This statement is used to create a table.

Syntax:

```
CREATE TABLE TableName (
Column1 datatype(size)[column_constraint],
Column2 datatype(size)[column_constraint],
....
ColumnN datatype(size)[column_constraint],
[table_constraint]
[,table_constraint]
);
```

Note: *The content in the square brackets indicates it is optional. If not required, you can skip it.*

Column constraints

- **PRIMARY KEY** // Use only, If one column name as primary key.
- **NOT NULL** // It does not accept NULL value in that column.
- **DEFAULT value** // It store default value in that column, if no value is inserted
- **UNIQUE** // It allows to store only unique values in the column

Table constraints

- **PRIMARY KEY(column_name1, column_name2, …)**
  Use it, If one column name or multiple column names acts as primary key.
- **UNIQUE(column_name1, column_name2, …)**
  Use it, if one column name or multiple column names should contain unique values.
  If multiple column names are used, then for each row, it consider values from all the columns mentioned to decide the uniqueness, but not column wise.
- **FOREIGN KEY** (column_name1) **REFERENCES** other_table_name (column_name2)
  It is used to link data from one table to other table.
- **CHECK(condition)**
  It does not allow inserting value(s), if the condition is not satisfied. The condition may also contain multiple column names.

Example 1: Creating table without any constraints

```
CREATE TABLE Employee_Info
(
   EmployeeID int,
   EmployeeName varchar(20),
   PhoneNumber numeric(10),
   City varchar(20),
   Country varchar(20)
);
```

Example 2: Using PRIMARY KEY and NOT NULL as column constraints

```
CREATE TABLE Departments
(
   DeptID int PRIMARY KEY,
   DeptName varchar(20)NOT NULL,
   Hod varchar(20),
   Location varchar(20)
);
```

Example 3: Using PRIMARY KEY, NOT NULL, UNIQUE and DEFAULT as column constraints and FOREIGN KEY as table constraint.

```
CREATE TABLE Students_Info
(
   HallTicketNo int PRIMARY KEY,
   Name varchar(20)NOT NULL,
   Mobile numeric(10)NOT NULL UNIQUE,
   DepartmentID int,
   City varchar(20)DEFAULT 'Hyderabad',
   FOREIGN KEY(DepartmentID) REFERENCES Departments (DeptID)
);
```

Example 4: Using NOT NULL, UNIQUE as column constraints and PRIMARY KEY and CHECK as table constraints.

```
CREATE TABLE Voter_list
(
   VoterID  numeric(10),
   AdhaarNo numeric(12)NOT NULL UNIQUE,
   Name varchar(20)NOT NULL,
   Age int,
   Mobile numeric(10) UNIQUE,
   City varchar(20),
   PRIMARY KEY(VoterID),
   CHECK(AGE>18)
);
```

c) The 'CREATE TABLE AS' Statement: You can also create a table from another existing table. The newly created table also contains data of existing table.

Syntax:

```
CREATE TABLE NewTableName AS(SELECT Column1,  column2, ..., ColumnN
FROM   ExistingTableName
WHERE  [condition]);
```

Example:
```
CREATE TABLE ExampleTable AS ( SELECT EmployeeName, PhoneNumber
                               FROM Employee_Info );
```

ii. DROP: This statement is used to drop an existing table or a database.

a) The 'DROP DATABASE' Statement: This statement is used to drop an existing database. When you use this statement, complete information present in the database will be lost.

Syntax:
```
DROP DATABASE DatabaseName;
```

Example:
```
DROP DATABASE Employee;
```

b) **The 'DROP TABLE' Statement:** This statement is used to drop an existing table. When you use this statement, complete information present in the table will be lost.

Syntax:
```
DROP TABLE TableName;
```

Example:
```
DROP TABLE Employee;
```

iii. TRUNCATE: This command is used to delete the information present in the table but does not delete the table. So, once you use this command, your information will be lost, but not the table.

Syntax:
```
TRUNCATE TABLE TableName;
```

Example:
```
TRUNCATE TABLE Employee_Info;
```

iv. ALTER: This command is used to add, delete or modify column(s) in an existing table. It can also be used to rename the existing table and also to rename the existing column name.

a) **The 'ALTER TABLE' with ADD column:** You can use this command to add a new column to the existing table.

Syntax:
```
ALTER TABLE TableName
 ADD ColumnName Datatype;
```

Example: Adding Blood Group column to the `Employee_Info` table

```
ALTER TABLE Employee_Info
ADD BloodGroup varchar(10);
```

b) **The 'ALTER TABLE' with DROP column:** You can use this command to remove a column from the existing table.

Syntax:
```
ALTER TABLE TableName
 DROP ColumnName;
```

Example: Removing Blood Group column from the `Employee_Info` table

```
ALTER TABLE Employee_Info
DROP BloodGroup;
```

c) **The 'ALTER TABLE' with MODIFY COLUMN:** This statement is used to change the data type or size of data type of an existing column in a table.

```
Syntax:    ALTER TABLE TableName
            MODIFY COLUMN ColumnName Datatype;
```

Example 1: Changing the size of column '*EmployeeName*' in table '*Employee_info*' from 20 to 30.

```
ALTER TABLE Employee_Info
MODIFY EmployeeName varchar(30);
```

Example 2: Changing the data type of column '*EmployeeID*' in the table '*Employee_info*' from *int* to *char(10)*.

```
ALTER TABLE Employee_Info
MODIFY EmployeeID  char(10);
```

d) **The 'ALTER TABLE' with CHANGE column name:** This statement is used to change the column name of an existing column in a table.

```
Syntax:    ALTER TABLE TableName
            CHANGE COLUMN OldColumnName  NewColumnName;
```

Example 1: Changing the column name '*EmployeeName*' to '*EmpName*' in table '*Employee_info*'.

```
ALTER TABLE Employee_Info
CHANGE COLUMN  EmployeeName EmpName;
```

e) **The 'ALTER TABLE' with RENAME table name:** This statement is used to change the table name in the database.

```
Syntax:    ALTER TABLE OldTableName
            RENAME TO NewTableName;
```

Example: Changing the table name from '*Employee_Info*' to '*Employee_Data*'.

```
ALTER TABLE Employee_Info
RENAME TO  Employee_Data;
```

3. DML COMMANDS:

The SQL commands that deals with the manipulation of data present in the database belong to DML or Data Manipulation Language and this includes most of the SQL statements. The DML commands are:

i. **INSERT:** This statement is used to insert new record (row) into the table.

Syntax:
```
INSERT INTO TableName[(Column1, Column2,..., ColumnN)]
 VALUES (value1, value2,..., valueN);
```

Example1 :

```
INSERT INTO Employee_Info( EmployeeID, EmployeeName, PhoneNumber, City,Country)
VALUES ('06', 'Sanjana', '9921321141', 'Chennai', 'India');
```

Example2 : When inserting all column values as per their order in the table, you can omit column names.

```
INSERT INTO Employee_Info
VALUES ('07', 'Sayantini','9934567654', 'Pune', 'India');
```

ii. **DELETE:** This statement is used to delete the existing records in a table.

Syntax:
```
DELETE FROM TableName
WHERE Condition;
```

Example:

```
DELETE FROM Employee_Info
WHERE EmployeeName='Preeti';
```

Note: *If where condition is not used in DELETE command, then all the rows data will be deleted. If used only rows which satisfies the condition are deleted*.

iii. **UPDATE:** This statement is used to modify the record values already present in the table.

Syntax:
```
UPDATE TableName
 SET Column1 = Value1, Column2 = Value2, ...
 [WHERE Condition];
```

Example:

```
UPDATE Employee_Info
SET EmployeeName = 'Jhon', City= 'Ahmedabad'
WHERE EmployeeID = 1;
```

Note: *If where condition is not used in UPDATE command, then in all the rows* `Employee Name changes to 'Jhon' and City name changes to 'Ahmedabad'`. *If used only rows which satisfies the condition are updated.*

4. DQL COMMAND: The purpose of DQL Command is to get data from one or more tables based on the query passed to it.

i. **SELECT:** This statement is used to select data from a database and the data returned is stored in a result table, called the **result-set**.

Syntax:

```
SELECT [DISTINCT] * / Column1,Column2,...ColumN
 FROM TableName
 [WHERE search_condition]
[GROUP BY column_names
 [HAVING search_condition_for_GROUP_BY]
 [ORDER BY column_name ASC/DESC] ;
```

Example 1:

```
SELECT * FROM table_name;
```

Example 2:

```
SELECT EmployeeID, EmployeeName
FROM Employee_Info;
```

The 'SELECT with DISTINCT' Statement: This statement is used to display only different unique values. It mean it will not display duplicate values.

Example :

```
SELECT DISTINCT PhoneNumber FROM Employee_Info;
```

The 'ORDER BY' Statement: The 'ORDER BY' statement is used to sort the required results in ascending or descending order. The results are sorted in ascending order by default. Yet, if you wish to get the required results in descending order, you have to use the **DESC** keyword.

Example

```
/* Select all employees from the 'Employee_Info' table sorted by
City  */

SELECT * FROM Employee_Info
ORDER BY City;

/*Select all employees from the 'Employee_Info' table sorted by
City in Descending order */

SELECT * FROM Employee_Info
ORDER BY City DESC;

/* Select all employees from the 'Employee_Info' table sorted by
City and EmployeeName. First it sort the rows as per city, then
sort by employee name */

SELECT * FROM Employee_Info
ORDER BY City, EmployeeName;

/* Select all employees from the 'Employee_Info' table sorted by
City in Descending order and EmployeeName in Ascending order: */

SELECT * FROM Employee_Info
ORDER BY City ASC, EmployeeName DESC;
```

AGGREGATE FUNCTIONS:

The SQL allows summarizing data through a set of functions called aggregate functions. The commonly used aggregate functions are: MIN( ), MAX( ), COUNT( ), SUM( ), AVG( ).

MIN() Function: The MIN function returns the smallest value of the selected column in a table.

Syntax:
```
SELECT MIN(ColumnName)
 FROM TableName
 WHERE Condition;
```

Example:
```
SELECT MIN(EmployeeID)
FROM Employee_Info;
```

MAX( ) Function: The MAX function returns the largest value of the selected column in a table.

Syntax:
```
SELECT MAX(ColumnName)
 FROM TableName
 WHERE Condition;
```

Example:
```
SELECT MAX(Salary) AS LargestFees
FROM Employee_Salary;
```

COUNT( ) Function: The COUNT function returns the number of rows which match the specified criteria.

Syntax:
```
SELECT COUNT(ColumnName)
 FROM TableName
 WHERE Condition;
```

Example:
```
SELECT COUNT(EmployeeID)
FROM Employee_Info;
```

SUM( ) Function: The SUM function returns the total sum of a numeric column that you choose.

Syntax:
```
SELECT SUM(ColumnName)
 FROM TableName
WHERE Condition;
```

Example:
```
SELECT SUM(Salary)
FROM Employee_Salary;
```

AVG( ) Function: The AVG function returns the average value of a numeric column that you choose.

Syntax:

```
SELECT AVG(ColumnName)
FROM TableName
WHERE Condition;
```

Example:

```
SELECT AVG(Salary)
FROM Employee_Salary;
```

The 'GROUP BY' Statement: This 'GROUP BY' statement is used with the aggregate functions to group the result-set by one or more columns.

Example:

```
-- To list the number of employees from each city.

SELECT COUNT(EmployeeID), City
FROM Employee_Info
GROUP BY City;
```

The 'HAVING' Clause: The 'HAVING' clause must be used SQL along with GROUP BY clause only. It is similar to the **WHERE** clause.

Example

```
/*&amp;nbsp;&amp;nbsp;To list the number of employees in each city. The employees
should be sorted high to low and only those cities must be included who have
more than 5 employees:*/

SELECT COUNT(EmployeeID), City
FROM Employee_Info
GROUP BY City
HAVING COUNT(EmployeeID) > 2;
```

Operators in SQL:

The different set of operators available in SQL are as follows:

- Arithmetic operators
- Bitwise operators
- Comparison operator
- Compound operator
- Logical operator

Let us look into each one of them, one by one.

Arithmetic Operators:

Operator	Description
%	Modulus [A % B]
/	Division [A / B]
*	Multiplication [A * B]
–	Subtraction [A – B]
+	Addition [A + B]

Bitwise Operators:

Operator	Description
^	Bitwise Exclusive OR (XOR) [A ^ B]
|	Bitwise OR [A | B]
&	Bitwise AND [A & B]

Comparison Operators:

Operator	Description
< >	Not Equal to [A < > B]
<=	Less than or equal to [A <= B]
>=	Greater than or equal to [A >= B]
<	Less than [A < B]
>	Greater than [A > B]
=	Equal to [A = B]

Compound Operators:

Operator	Description
|*=	Bitwise OR equals [A |*= B]
^-=	Bitwise Exclusive equals [A ^-= B]
&=	Bitwise AND equals [A &= B]
%=	Modulo equals [A %= B]
/=	Divide equals [A /= B]
*=	Multiply equals [A*= B]
-=	Subtract equals [A-= B]
+=	Add equals [A+= B]

Logical Operators: The Logical operators present in SQL are as follows: AND, OR, NOT, BETWEEN, LIKE, IN, EXISTS, ALL, ANY.

AND Operator: This operator is used to filter records that rely on more than one condition. This operator displays the records, which satisfy all the conditions separated by AND, and give the output TRUE.

Syntax:

```
SELECT Column1, Column2, ..., ColumnN
FROM TableName
WHERE Condition1 AND Condition2 AND Condition3 ...;
```

Example:

```
SELECT * FROM Employee_Info
WHERE City='Mumbai' AND City='Hyderabad';</pre>
```

OR Operator: This operator displays all those records which satisfy any of the conditions separated by OR and give the output TRUE.

Syntax:
```
SELECT Column1, Column2, ..., ColumnN
FROM TableName
WHERE Condition1 OR Condition2 OR Condition3 ...;
```

Example:

```
SELECT * FROM Employee_Info
WHERE City='Mumbai' OR City='Hyderabad';
```

NOT Operator: The NOT operator is used, when you want to display the records which do not satisfy a condition.

Syntax:
```
SELECT Column1, Column2, ..., ColumnN
 FROM TableName
 WHERE NOT Condition;
```

Example:

```
SELECT * FROM Employee_Info
WHERE NOT City='Mumbai';
```

NOTE: You can also combine the above three operators and write a query as follows:

```
SELECT * FROM Employee_Info
WHERE NOT Country='India' AND (City='Bangalore' OR City='Hyderabad');
```

BETWEEN Operator: The BETWEEN operator is used, when you want to select values within a given range. Since this is an inclusive operator, both the starting and ending values are considered.

Syntax:
```
SELECT ColumnName(s)
  FROM TableName
  WHERE ColumnName BETWEEN Value1 AND Value2;
```

Example:

```
SELECT * FROM Employee_Salary
WHERE Salary BETWEEN 40000 AND 50000;
```

LIKE Operator

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column of a table. There are mainly two wildcards that are used in conjunction with the LIKE operator:

- **%** : It is used to matches 0 or more character.
- **_** : It is used to matches exactly one character.

Syntax

```
SELECT ColumnName(s)
FROM TableName
WHERE ColumnName LIKE pattern;
```

Refer to the following table for the various patterns that you can mention with the LIKE operator.

Like Operator Condition	Description
WHERE CustomerName LIKE 'v%	Finds any values that start with "v"
WHERE CustomerName LIKE '%v'	Finds any values that end with "v"
WHERE CustomerName LIKE '%and%'	Finds any values that have "and" in any position
WHERE CustomerName LIKE '_q%'	Finds any values that have "q" in the second position.
WHERE CustomerName LIKE 'u_%_%'	Finds any values that start with "u" and are at least 3 characters in length
WHERE ContactName LIKE 'm%a'	Finds any values that start with "m" and end with "a"

Example:

```
SELECT * FROM Employee_Info
WHERE EmployeeName LIKE 'S%';
```

IN Operator: This operator is used for multiple OR conditions. This allows you to specify multiple values in a WHERE clause.

Syntax:

```
SELECT ColumnName(s)
 FROM TableName
 WHERE ColumnName IN (Value1,Value2...);
```

Example:

```
SELECT * FROM Employee_Info
WHERE City IN ('Mumbai', 'Bangalore', 'Hyderabad');
```

NOTE: You can also use IN while writing Nested Queries.

EXISTS Operator: The EXISTS operator is used to test if a record exists or not.

Syntax:
```
SELECT ColumnName(s)
 FROM TableName
 WHERE EXISTS
 (SELECT ColumnName FROM TableName WHERE condition);
```

Example:
```
SELECT City
FROM Employee_Info
WHERE EXISTS (SELECT City
              FROM Employee_Info
              WHERE EmployeeId = 05 AND City = 'Kolkata');
```

ALL Operator: The ALL operator is used with a WHERE or HAVING clause and returns TRUE if all of the subquery values meet the condition.

Syntax:
```
SELECT ColumnName(s)
 FROM TableName
 WHERE ColumnName operator ALL
        (SELECT ColumnName FROM TableName WHERE condition);
```

Example:
```
SELECT EmployeeName
FROM Employee_Info
WHERE EmployeeID = ALL ( SELECT EmployeeID
                         FROM Employee_Info
                         WHERE City = 'Hyderabad');
```

ANY Operator: Similar to the ALL operator, the ANY operator is also used with a WHERE or HAVING clause and returns true if any of the subquery values meet the condition.

Syntax:
```
SELECT ColumnName(s)
 FROM TableName
 WHERE ColumnName operator ANY
 (SELECT ColumnName FROM TableName WHERE condition);
```

Example:
```
SELECT EmployeeName
FROM Employee_Info
WHERE EmployeeID = ANY (SELECT EmployeeID
                        FROM Employee_Info
                        WHERE City = 'Hyderabad' OR City = 'Kolkata');
```

Aliases Statement: Aliases are used to give a column / table a temporary name and only exists for duration of the query.

Syntax:
```
/* Alias Column Syntax. Instead of displaying the column name
 used in the table, it display alias name. */

 SELECT ColumnName AS AliasName
 FROM TableName;
```

Example:

```
SELECT EmployeeID AS ID, EmployeeName AS EmpName
FROM Employee_Info;
```

5. NESTED QUERIES

Nested queries are those queries which have an outer query and inner subquery. So, basically, the subquery is a query which is nested within another query.

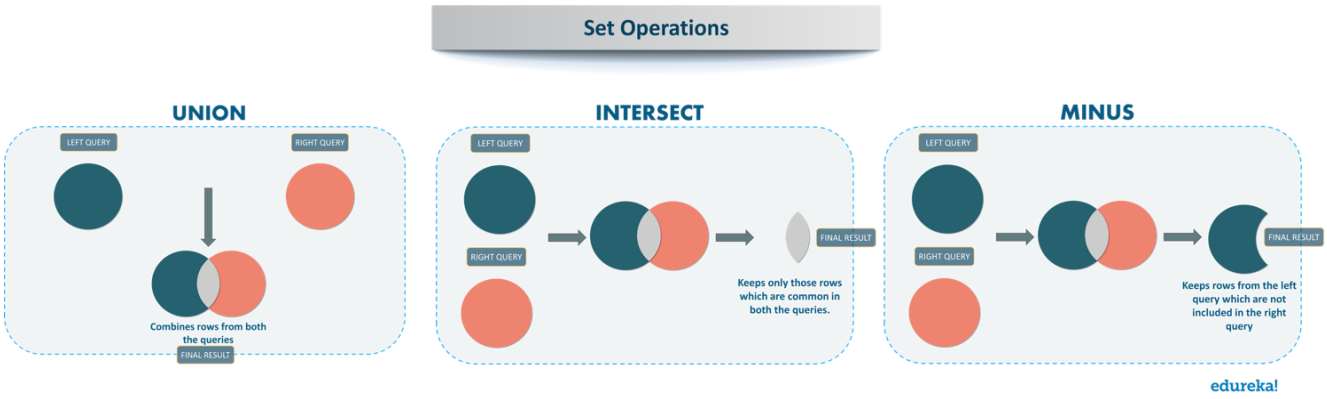
OUTER QUERY *SUBQUERY or INNER QUERY*

```
SELECT EmployeeName, PhoneNumber
FROM Employee_Info
WHERE City IN ( SELECT City
                FROM Office
                WHERE County = 'INDIA') ;
```

First the inner query gets executed and the result will be used to execute the outer query.

6. SET OPERATIONS: UNION, INTERSECT, EXCEPT

There are mainly three set operations: UNION, INTERSECT, EXCEPT. You can refer to the image below to understand the set operations in SQL.



i. **UNION:** This operator is used to combine the result-set of two or more SELECT statements.

Syntax:

```
SELECT ColumnName(s) FROM Table1 WHERE condition
   UNION
SELECT ColumnName(s) FROM Table2 WHERE condition;
```

ii. **INTERSECT:** This clause used to combine two SELECT statements and return the intersection of the data-sets of both the SELECT statements.

Syntax:
```
SELECT ColumnName(s) FROM Table1 WHERE condition
   INTERSECT
 SELECT ColumnName(s) FROM Table2 WHERE condition;
```

iii. **EXCEPT:** This operator returns those tuples that are returned by the first SELECT operation, and are not returned by the second SELECT operation.

Syntax:
```
SELECT ColumnName(s) FROM Table1 WHERE condition
   EXCEPT
 SELECT ColumnName(s) FROM Table2 WHERE condition;
```

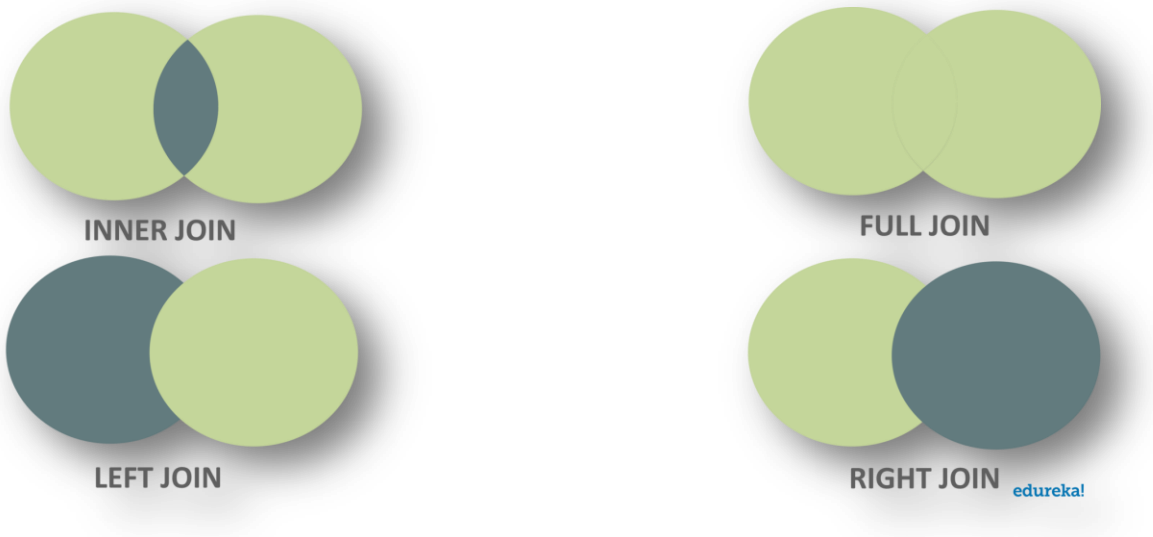
Note: UNION, INTERSECT or EXCEPT operations are possible if and only if first SELECT query and second SELECT query produces same no of columns in same order, same column names and data type. Otherwise it gives an error.

7. JOINS

JOINS are used to combine rows from two or more tables, based on a related column between those tables. The following are the types of joins:

- **INNER JOIN:** This join returns those records which have matching values in both the tables.
- **FULL JOIN:** This join returns all those records which either have a match in the left or the right table.
- **LEFT JOIN:** This join returns records from the left table, and also those records which satisfy the condition from the right table.
- **RIGHT JOIN:** This join returns records from the right table, and also those records which satisfy the condition from the left table.

Refer to the image below.



Let's consider the below `Technologies` and the `Employee_Info` table, to understand the syntax of joins.

Employee_Info

EmployeeID	EmployeeName	PhoneNumber	City	Country
01	Shravya	9898765612	Mumbai	India
02	Vijay	9432156783	Delhi	India
03	Preeti	9764234519	Bangalore	India
04	Vijay	9966442211	Hyderabad	India
05	Manasa	9543176246	Kolkata	India

Technologies

TechID	EmpID	TechName	ProjectStartDate
1	01	DevOps	04-01-2019
2	03	Blockchain	06-07-2019
3	04	Python	01-03-2019
4	06	Java	10-10-2019

INNER JOIN or EQUI JOIN: This is a simple JOIN in which the result is based on matched data as per the equality condition specified in the SQL query. This join is used mostly. NATURAL JOIN is a type INNER JOIN. We can also use it. It also gives same result.

Syntax

```
SELECT ColumnName(s)
FROM Table1
INNER JOIN Table2 ON Table1.ColumnName = Table2.ColumnName;
```

Example

```
SELECT T.TechID, E.EmployeeID, E.EmployeeName
FROM Technologies T
INNER JOIN Employee_Info E ON T.EmpID = E.EmpID;
```

TechID	EmployeeID	EmployeeName
1	01	Shravya
2	03	Preeti
3	04	Vijay

FULL OUTER JOIN: The full outer join returns a result-set table with the **matched data** of two table then remaining rows of both **left** table and r**ight** table with missing values are filled with NULL values.

Syntax

```
SELECT ColumnName(s)
FROM Table1
FULL OUTER JOIN Table2 ON Table1.ColumnName = Table2.ColumnName;
```

Example

```
SELECT E.EmployeeID, E.EmployeeName, T.TechID
FROM Employee_Info E
FULL OUTER JOIN Technologies T ON E.EmployeeID=T.EmployeeID;
```

EmployeeID	EmployeeName	TechID
01	Shravya	1
02	Vijay	NULL
03	Preeti	2
04	Vijay	3
05	Manasa	NULL
06	NULL	4

LEFT JOIN: The left outer join returns a result-set table with the **matched data** from the two tables and then the remaining rows of the **left** table with null for the **right** table's columns.

Syntax:

```
SELECT ColumnName(s)
FROM Table1
LEFT JOIN Table2 ON Table1.ColumnName = Table2.ColumnName;
```

Example:

```
SELECT E.EmployeeId, E.EmployeeName, T.TechID
FROM Employee_Info E
LEFT JOIN Technologies T ON E.EmployeeID = T.EmpIDID ;
```

EmployeeID	EmployeeName	TechID
01	Shravya	1
02	Vijay	NULL
03	Preeti	2
04	Vijay	3
05	Manasa	NULL

RIGHT JOIN: The right outer join returns a result-set table with the matched data from the two tables being joined, then the remaining rows of the right table and null for the remaining left table's columns.

Syntax:

```
SELECT ColumnName(s)
FROM Table1
RIGHT JOIN Table2 ON Table1.ColumnName = Table2.ColumnName;
```

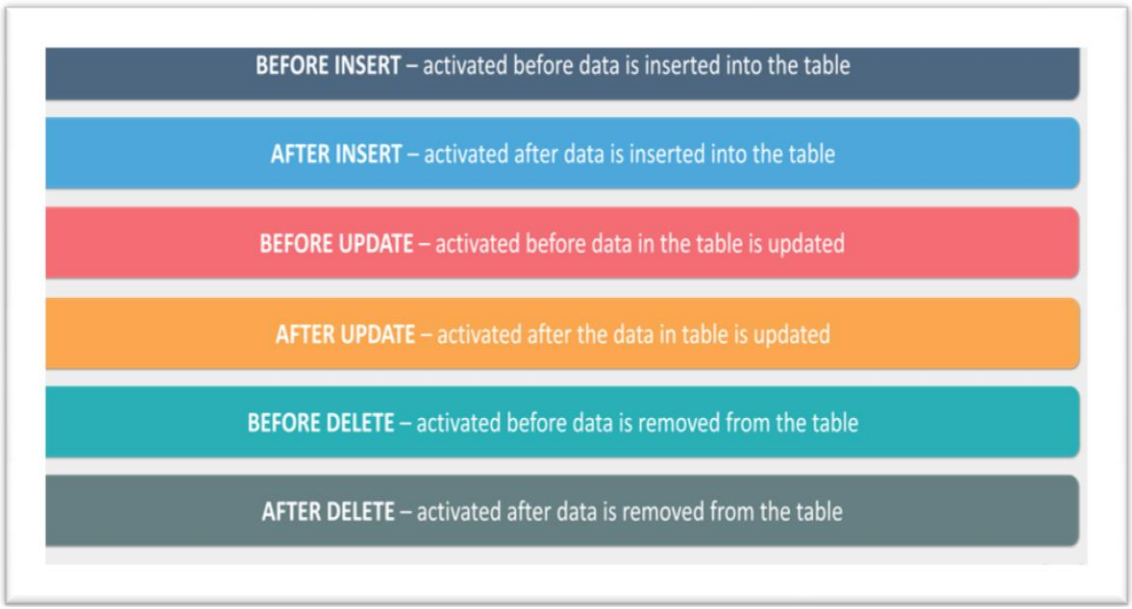
Example:

```
SELECT E.EmployeeId, E.EmployeeName, T.TechID
FROM Employee_Info E
RIGHT JOIN Technologies T ON E.EmployeeID = T.EmpIDID ;
```

EmployeeID	EmployeeName	TechID
01	Shravya	1
03	Preeti	2
04	Vijay	3
NULL	NULL	4

8. TRIGGERS

A trigger is a stored procedure in database which automatically invokes whenever a special event in the database occurs. For example, a trigger can be invoked when a row is inserted into a specified table or when certain table columns are being updated. So, a **trigger** can be invoked either **BEFORE** or **AFTER** the data is changed by **INSERT**, **UPDATE** or **DELETE** statement. Refer to the image below.



Syntax:

```
CREATE TRIGGER [TriggerName]
[BEFORE | AFTER]
{INSERT | UPDATE | DELETE}
on [TableName]
[FOR EACH ROW]
[TriggerBody]
```

Explanation of syntax:

- create trigger [trigger_name]: Creates or replaces an existing trigger with the trigger_name.
- [before | after]: This specifies when the trigger will be executed.
- {insert | update | delete}: This specifies the DML operation.
- on [table_name]: This specifies the name of the table associated with the trigger.
- [for each row]: This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected.
- [trigger_body]: This provides the operation to be performed as trigger is fired .

BEFORE and AFTER of Trigger:

BEFORE triggers run the trigger action before the triggering statement is run.

AFTER triggers run the trigger action after the triggering statement is run.

EXAMPLE:

```
CREATE TRIGGER nb BEFORE INSERT ON accounts FOR EACH ROW      /* Event */
Begin
  IF :NEW.bal < 0  THEN                                    /*Condition*/
    DBMS_OUTPUT.PUT_LINE('BALANCE IS NAGATIVE..');          /*Action*/
  END IF;
End;
```

A trigger called '*nb*' is created to alert the user when inserting account details with negative balance value in to accounts table. Before inserting, the trigger is activated if the condition is true. When a trigger activated, the action part of the trigger is get executed.

9. NORMALIZATION

- Normalization is the process of minimizing the redundancy from a relation or set of relations.
- It is used to eliminate the Insertion, Update and Deletion Anomalies.
- Normalization divides the larger table into the smaller table and links them using relationship.
- Normalization is done with the help of different normal form.

The inventor of the relational model Edgar Codd proposed the theory of normalization with the introduction of the First Normal Form, and he continued to extend theory with Second and Third Normal Form. Later he joined Raymond F. Boyce to develop the theory of Boyce-Codd Normal Form. In software industry, they are using only up to third normal form and sometimes Boyce-Codd Normal Form.

The Problem of redundancy

Redundancy means having multiple copies of same data in the database. This problem arises when a database is not normalized. Redundancy leads the following problems.

- **Wastage of Memory:** Disk space is wasted due to storing same copy multiple times.
- **Storage cost increases:** When multiple copies of same data is stored, need more disk space and storage cost increases.
- **Update anomaly:** When Address of student is stored at several places; a change in the address must be made in all the places. Changing the address at some places and leaving other places leads to inconsistency problem.
- **Insertion Anomaly:** The nature of a database may be such that it is not possible to add a required piece of data unless another piece of unavailable data is also added. For example, a library database cannot store the details of a new student until that student has taken atleast one book from the library.
- **Deletion Anomaly:** When some data is deleted, it also deletes other data automatically. For example, deleting a book details from a library database, it also delete the student details who have taken the book previously.

10. 1NF (FIRST NORMAL FORM)

A relation (table) is said to be in first normal form if and only if:

- Each table cell contains only atomic values (single value).
- Each record needs to be uniquely identified by the primary key.

1NF Example:

HTNO	FIRST NAME	LAST NAME	MOBILE
501	Jhansi	Rani	9999988888 7777799999
502	Ajay	Kumar	8888888881 7897897897
503	Priya	Verma	9898989898

The above table is not in 1NF because **501** and **502** is having two values in mobile column. If we add a new column as *alternative mobile number* to the above table, then for 503 *alternative mobile number* is NULL. Moreover, if a student has 'n' mobile numbers, then adding 'n' extra column is meaningless. It is better to add extra rows. If we add extra row for each 501 and 502 then the table looks like

HTNO	FIRST NAME	LAST NAME	MOBILE
501	Jhansi	Rani	9999988888
501	Jhansi	Rani	7777799999
502	Ajay	Kumar	8888888881
502	Ajay	Kumar	7897897897
503	Priya	Verma	9898989898

But the above table violates primary key constraint. Therefore instead of adding either columns or rows, the best solution is to split the table into two tables as shown below. If we do as shown below, if a student having 'n' number of mobile numbers also can be added.

HTNO	FIRST NAME	LAST NAME
501	Jhansi	Rani
502	Ajay	Kumar
503	Priya	Verma

HTNO	MOBILE
501	9999988888
501	7777799999
502	8888888881
502	7897897897
503	9898989898

11. 2NF (SECOND NORMAL FORM)

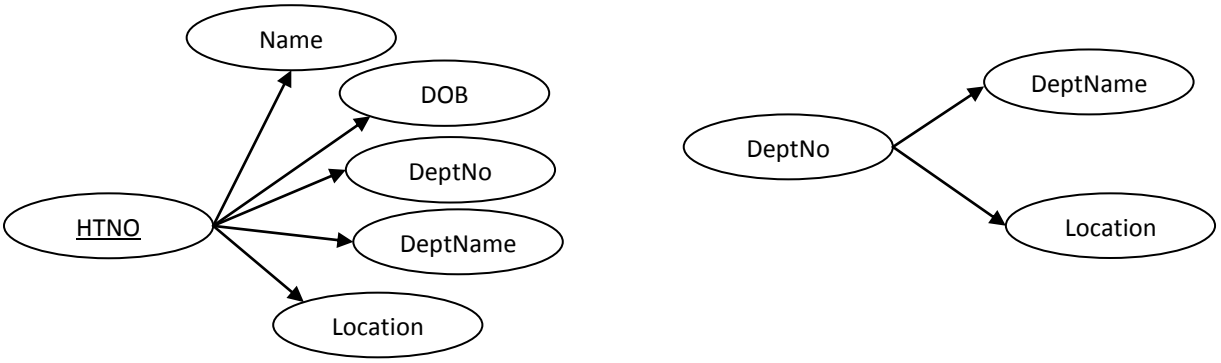
A relation is said to be in 2-NF if and only if

- It should be in 1-NF (First Normal Form)
- There should not be any partial functional dependencies

2NF Example:

HTNO	*Name*	*DOB*	*DeptNo*	*DeptName*	*Location*
501	Jhansi	30-10-1998	05	CSE	A-Block
502	Ajay	24-12-1999	05	CSE	A-Block
410	Priya	12-03-2000	04	ECE	B-Block
120	Rahul	30-10-1998	01	CIVIL	C-Block
415	Smitha	18-06-1999	04	ECE	B-Block

The above table is not in 2NF because there exist partial function dependencies. ***HTNO*** is a key attribute in the above table. If every non-key attribute fully dependent on key attribute, then we say it is fully functional dependent. Consider the below diagram. ***{Name, DOB, DeptNo, DeptName, Location}*** depends on ***HTNO***. But ***{DeptName, Location}*** also depends on ***DeptNo***.



It is clear that ***DeptName*** and ***Location*** not only depends upon ***HTNO*** but also on ***DeptNo***. So, there exists partial function dependency. This partial functional dependency can be removed by splitting the above table into two tables as follows.

HTNO	*Name*	*DOB*	*DeptNo*
501	Jhansi	30-10-1998	05
502	Ajay	24-12-1999	05
410	Priya	12-03-2000	04
120	Rahul	30-10-1998	01
415	Smitha	18-06-1999	04

DeptNo	**DeptName**	**Location**
05	CSE	A-Block
04	ECE	B-Block
01	CIVIL	C-Block

12. 3NF (THIRD NORMAL FORM)

A relation (table) is in third normal form if and only if it satisfies the following conditions:

- It is in second normal form
- There is no transitive functional dependency

Transitive functional dependency means, we have the following relationships in the table: A is functionally dependent on B (A→B), and B is functionally dependent on C (B→C). In this case, C is transitively dependent on A via B (A→B and B→C mean A→B→C implies A→C).

3NF Example:

Consider the following book details table example:

BOOK_DETAILS

BookID	GenreID	GenreType	Price
1	1	Gardening	250.00
2	2	Sports	149.00
3	1	Gardening	100.00
4	3	Travel	160.00
5	2	Sports	320.00

The above table is not in 3NF because there exist transitive dependency. In the table able,

BookID determines ***GenreID*** ***{ BookID → GenreID }***

GenreID determines ***GenreType***. ***{ GenreID → GenreType }***

∴ ***BookID*** determines ***GenreType*** via ***GenreID.*** { ***BookID → GenreType*** }

It implies that transitive functional dependency is existing and the structure does not satisfy third normal form. To bring this table in to third normal form, we split the table into two as follows:

BOOK_DETAILS

BookID	GenreID	Price
1	1	250.00
2	2	149.00
3	1	100.00
4	3	160.00
5	2	320.00

GENRE_DETAILS

GenreID	GenreType
1	Gardening
2	Sports
3	Travel

13. *B*OYCE *C*ODD *N*ORMAL *F*ORM (BCNF)

A relation (table) is said to be in the BCNF if and only if it satisfy the following conditions:

- It should be in the **Third Normal Form**.
- For any functional dependency A → B, A should be a **super key**.

In simple words, it means, that for a dependency A → B, A cannot be a **non-prime attribute**, if B is a **prime attribute**.

Example: Below we have a Patient table of a hospital. A patient can go to hospital many times to take treatment. On a single day many patients can take treatment.

PatientID	Name	EmailID	AdmittedDate	Drug	Quntity
101	Ram	ram@gmail.com	30/10/1998	A-10	10
102	Jhon	jho@gmail.com	30/10/1998	X-90	10
101	Ram	ram@gmail.com	10/06/2001	X-90	20
103	Sowmya	sam@gmail.com	05/03/2002	Y-30	15
102	Jhon	jho@gmail.com	05/03/2002	A-10	15

In the above table, {***PateintID, AdmittedDate***} acts as Primary key. But if we know the ***EmailID*** value, we can find ***PatientID*** value.

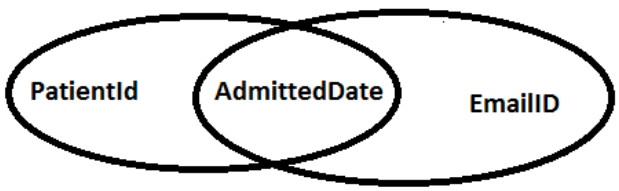
That is ***EmailID*** → ***PatientID***.

In the above dependency, ***EmailId*** is non-prime attribute and ***PatientID*** is a prime attribute. Therefore the above table is not in BCNF. In order to bring the table into BCNF, we split it into two tables as shown below.

PatientID	*Name*	*AdmittedDate*	*Drug*	*Quntity*
101	Ram	30/10/1998	A-10	10
102	Jhon	30/10/1998	X-90	10
101	Ram	10/06/2001	X-90	20
103	Sowmya	05/03/2002	Y-30	15
102	Jhon	05/03/2002	A-10	15

PatientID	*EmailID*
101	ram@gmail.com
102	jho@gmail.com
103	sam@gmail.com

In other words we can also define BCNF as there should not be any overlapping between candidate keys. If you consider the original table (before splitting), we can get two candidate keys {***PateintID, AdmittedDate***} and {***EmailID, AdmittedDate***}.



As there exist overlapping in the candidate keys, the table is not in BCNF. To bring it into BCNF, we split into two tables as shown above.

14. 4-NF (FOURTH NORMAL FORM)

A relation is said to be in 4-NF if and only if it satisfies the following conditions

- It should be in the **Third Normal Form**.
- The table should not have any **Multi-valued Dependency**.

What is Multi-valued Dependency?

A table is said to have multi-valued dependency, if the following three conditions are true.

i. A table should have at-least 3 columns for it to have a multi-valued dependency.

ii. For any dependency $A \rightarrow B$, if there exists multiple value of B for a single value of A, then the table may have multi-valued dependency. It is represented as $A \rightarrow\rightarrow B$.

iii. In a relation `R(A,B,C)`, if there is a multi-valued dependency between A and B, then B and C should be independent of each other.

If all these three conditions are true for any relation (table), then it contains multi-valued dependency. The multi-valued dependency can be explained with an example. Let the Relation R containing three columns *A, B, C* and four rows *s, t, u, v.*

	A	B	C
s	a1	b1	c1
t	a1	b1	c2
u	a1	b2	c1
v	a1	b2	c2

If s(A) = t(A) = u(A) = v(A)

s(B) = t(B) and s(B) = v(B)

s(C) = u(C) and t(C) = v(C) , then there exist multi-valued dependency.

Example: Consider the below `college enrolment` table with columns `HTNO`, `Subject` and `Hobby`.

As shown in the above figure, if 501 opted for subjects like Java and C# and hobbies of 501 are Cricket and Dancing. Similarly, If 502 opted for subjects like Python and Android and hobbies of 501 are Chess and Singing, then it can be written into a table with three columns as follows:

HTNO	***Subject***	***Hobby***
501	Java	Cricket
501	Java	Dancing
501	C#	Cricket
501	C#	Dancing
502	Python	Chess
502	Python	Singing
502	Android	Chess
502	Android	Singing

As there exist multi valued dependency, the above table is decomposed into two tables such that

HTNO	***Subject***
501	Java
501	C#
502	Python
502	Android

HTNO	***Hobby***
501	Cricket
501	Dancing
502	Chess
502	Singing

Now these tables (relations) satisfy the fourth normal form.

15. 5NF

A relation is said to be in 5-NF if and only if it satisfies the following conditions

- It should be in the **Fourth Normal Form**.
- The table should not have any **join Dependency** and joining should be lossless.

5NF is also known as Project-join normal form (PJ/NF).

A table is decomposed into multiple small tables to eliminate redundancy, and when we re-join the decomposed tables, there should not be any loss in the original data or shold not create any new data. In simple words, joining two or more decomposed table should not lose records nor create new records.

Example: Consider a table which contains a record of **Subject**, **Professor** and **Semester** in three columns. The primary key is the combination of all three columns. No column itself is not a candidate key or a super key.

In the table, DBMS is taught by Ravindar and Uma Rani in semester 4, DS by Sindhusha and Venu in sem 3. In this case, the combination of all these fields required to identify valid data.

So to make the table into 5NF, we can decompose it into three relations,

Subject	***Professor***	***Semester***
C	Srilatha	2
DBMS	Ravindar	4
DS	Sindhusha	3
DBMS	Uma Rani	4
CN	Srikanth	5
DS	Venu	3
WT	Srinivas	5

The above table is decomposed into three tables as follows to bring it into 5-NF.

Subject	***Professor***
C	Srilatha
DBMS	Ravindar
DS	Sindhusha
DBMS	Uma Rani
CN	Srikanth
DS	Venu
WT	Srinivas

Semester	***Professor***
2	Srilatha
3	Ravindar
5	Sindhusha
3	Uma Rani
5	Srikanth
2	Venu
5	Srinivas

Semester	***Subject***
2	C
4	DBMS
3	DS
5	CN
5	WT

16. LOSS LESS JOIN DECOMPOSITION

Decomposition of a relation R into R1 and R2 is lossless-join decomposition if at least one of the following functional dependencies are in F+ (Closure of functional dependencies)

$$R1 \cap R2 \rightarrow R1$$
$$\text{OR}$$
$$R1 \cap R2 \rightarrow R2$$

- Consider a relation R which is decomposed into sub relations R_1 and R_2 .
- This decomposition is called lossless join decomposition when we join R_1 and R_2 and if we get the same relation R that was decomposed.
- For lossless join decomposition, we always have: $\mathbf{R_1 \bowtie R_2}$

Example 1: Consider the following relation R(A , B , C). Let this relation is decomposed into two sub relations R_1(A , B) and R_2(B , C)

R

A	B	C
1	2	1
2	5	3
3	3	3

decompose →

R_1

A	B
1	2
2	5
3	3

and

R_2

B	C
2	1
5	3
3	3

Now, let us check whether this decomposition is lossless or not. For lossless decomposition, we must have: $\mathbf{R_1 \bowtie R_2 = R}$. Now, if we perform the natural join ($\bowtie$) of the sub relations R_1 and R_2 , we get

A	B	C
1	2	1
2	5	3
3	3	3

This relation is same as the original relation R.

Thus, we conclude that the above decomposition is lossless join decomposition. This is because the resultant relation after joining the sub relations is same as the decomposed relation. No extraneous tuples (rows) appear after joining of the sub-relations.

Example 2: Consider the following relation R(A , B , C). Let this relation is decomposed into two sub relations R_1(A , C) and R_2(B , C)

R

A	B	C
1	2	1
2	5	3
3	3	3

decompose →

R_1

A	C
1	1
2	3
3	3

and

R_2

B	C
2	1
5	3
3	3

Now, let us check whether this decomposition is lossless or not. For lossless decomposition, we must have: $\mathbf{R_1 \bowtie R_2 = R}$. Now, if we perform the natural join ($\bowtie$) of the sub relations R_1 and R_2 , we get

A	B	C
1	2	1
2	5	3
2	3	3
3	5	3
3	3	3

This relation is not same as the original relation R.

Thus, we conclude that the above decomposition is **not** lossless join decomposition. This is because the resultant relation after joining the sub relations is **not** same as the decomposed relation. Extraneous tuples (rows) appear after joining of the sub-relations.

Determine whether the decomposition of R into R_1 (A , B) , R_2 (B , C) and R_3 (B , D) is lossless or lossy.

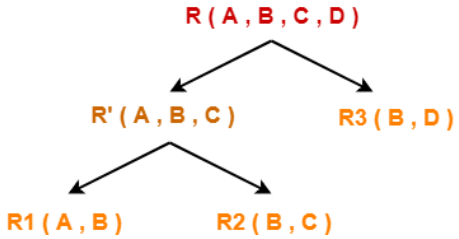
Solution:

Strategy to Solve: When a given relation is decomposed into more than two sub relations, then

- Consider any one possible ways in which the relation might have been decomposed into those sub relations.
- First, divide the given relation into two sub relations.
- Then, divide the sub relations according to the sub relations given in the question.

As a thumb rule, remember-
Any relation can be decomposed only into two sub relations at a time.

Consider the original relation R was decomposed into the given sub relations as shown:



Decomposition of R(A, B, C, D) into R'(A, B, C) and R_3(B, D)-

To determine whether the decomposition is lossless or lossy,

- We will check all the conditions one by one.
- If any of the conditions fail, then the decomposition is lossy otherwise lossless.

Condition-01: According to condition-01, union of both the sub relations must contain all the attributes of relation R. So, we have

R‘ (A , B , C) ∪ R_3 (B , D) = R (A , B , C , D)

Clearly, union of the sub relations contains all the attributes of relation R. Thus, condition-01 satisfies.

Condition-02: According to condition-02, intersection of both the sub relations must not be null. So, we have

R‘ (A , B , C) ∩ R_3 (B , D) = B

Clearly, intersection of the sub relations is not null. Thus, condition-02 satisfies.

Condition-03: According to condition-03, intersection of both the sub relations must be the super key of one of the two sub relations or both. So, we have-

R‘ (A , B , C) ∩ R_3 (B , D) = B