Shiksha Mandal's

G. S. College of Commerce, Wardha

DEPARTMENT OF B. COM. COMPUTER APPLICATION

BCCA Part-II

SEM-III

Database Management System

Unit –IV

By

Prof. Harsha N. Gangavane

Dr. Revati Bangre
Co-ordinator

Dr. Anil Ramteke
Principal (Officiating)

Data Constraints

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.

These constraints have already been discussed in SQL - RDBMS Concepts but it's worth to revise them at this point.

Types of SQL Constraints:

- NOT NULL - Ensures that a column cannot have a NULL value.
- UNIQUE - Ensures that all values in a column are **different**.
- PRIMARY KEY - A combination of a NOT NULL and UNIQUE. ...
- FOREIGN KEY - Uniquely identifies a row/record in **another table**.
- CHECK - Ensures that all values in a column satisfies a specific condition.
- DEFAULT - Constraint is used to provide a default value for the fields of a table.

- **NOT NULL Constraint**

By default, a column can hold NULL values. If we don't enter any value in a particular column, it will store NULL if the default value has not been specified. The NULL value is omitted in various operations in a database. To ensure that a column does not contain a NULL value, ***NOT NULL*** constraint is used.

For example:

```
CREATE TABLE Employees
(
 Emp_Id INT(3) NOT NULL,
 Emp_Name VARCHAR(30) NOT NULL,
 Salary DOUBLE(8,2)
);
```

This query will create a table Employees with fields Emp_Id, Emp_Name, and Salary. Emp_Id and Emp_Name are specified with NOT NULL constraint that means they cannot contain NULL value **while** salary may contain NULL value.

If Employees table has already been created, then to add a NOT NULL constraint to the Emp_Id and Emp_name column use following syntax:

```
ALTER TABLE Employees MODIFY Emp_Id INT(3) NOT NULL;
OR
ALTER TABLE Employees ADD [CONSTRAINT notNullEmp] NOT NULL(Emp_Id);
```

```
INSERT INTO Employees(Emp_Id,Emp_Name) VALUES(001,”John Doe”);
```

This statement will store values in specified fields and store NULL in salary field. As Salary field doesnot have a NOT NULL constraint, thus no error will be generated.

```
INSERT INTO Employees(Emp_Id,Salary) VALUES(007,200000);
```

This statemnt will store NULL in Emp_Name as no value has been assigned to it. But Emp_Name field has a NOT NULL constraint thus an error will be generated as NULL value cannot be ass

NOT NULL constraint applied to a column ensures that the particular column always contains a value, i.e., we cannot insert a new record, or update a record without providing a value for that field.

- **UNIQUE Constraint**

UNIQUE constraint ensures that all the values stored in a column are different from each other. The UNIQUE Constraint prevents two records from having identical values in a column, i.e., the values must not be repeated. We can use UNIQUE constraint in a single column or multiple columns.

```
CREATE TABLE Employees
(
 Emp_Id INT(3) UNIQUE,
 Emp_Name VARCHAR(30),
 Salary DOUBLE(8,2)
);
```

This query will create a table **Employees(Emp_Id, Emp_Name and Salary).** The Emp_Id has a UNIQUE constraint thus each row of Emp_Id field will have a different value.

If the Employees table has already been created, then to add a UNIQUE constraint to the Emp_Id column:

```
ALTER TABLE Employees MODIFY Emp_Id INT(3) UNIQUE;
OR
ALTER TABLE Employees ADD [CONSTRAINT myUniqueConstraint] UNIQUE(Emp_Id);
```

To drop a UNIQUE constraint, use the following SQL query:

```
ALTER TABLE Employees DROP [UNIQUE Emp_Id | CONSTRAINT myUniqueConstraint];
```

- **PRIMARY KEY Constraint**

A primary key constraint uniquely identifies each row/record in a database table. Primary keys must contain unique values. A primary key column cannot have NULL values.

A table can have only one primary key, which may consist of single or multiple fields. When multiple fields are used as a primary key, they are called a composite key. The field with a PRIMARY KEY constraint can be used to uniquely identify each record of a table in various database operations.

For example:

```
CREATE TABLE Employees
 (
 Emp_Id INT(3) PRIMARY KEY,
 Emp_Name VARCHAR(30),
 Salary DOUBLE(8,2)
);
```

This query will create a table **Employees** (Emp_Id, Emp_Name and Salary) with Emp_Id field as PRIMARY KEY constraint.

<u>To create a PRIMARY KEY constraint</u> on the "EMP_ID" column when Employees table already exists, use the following SQL syntax:

```
        ALTER TABLE Employees MODIFY Emp_Id INT(3) PRIMARY KEY;
```

OR

```
        ALTER TABLE Employees ADD [CONSTRAINT PK_Emp_Id] PRIMARY KEY
(Emp_Id);
```

<u>To drop PRIMARY KEY Constraints from a table :</u>

```
        ALTER TABLE Employees DROP [PRIMARY KEY | CONSTRAINT PK_Emp_Id];
```

- **FOREIGN KEY Constraint**

Foreign Key Constraint is used to link two tables. It is used to establish a relationship between the data in two tables. A Foreign Key is a column or a combination of columns whose values match a Primary Key in a different table. The table containing the FOREIGN KEY is called the child table, and the table containing the candidate key is called the parent table.

FOREIGN KEY is used to enforce referential integrity. It is used to prevent actions that would destroy links between tables. It also prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the table it points to.

```
      CREATE TABLE Employees
       (
       Emp_Id INT(3) PRIMARY KEY,
       Emp_Name VARCHAR(30),
       Dept_Id INT(5) REFERENCES Department(Dept_Id)
      );
```

This query creates a table **Employees**(Emp_Id, Emp_Name and Dept_Id) with Dept_Id as FOREIGN KEY which links Employees table to Department table. To create a FOREIGN KEY constraint on the "Dept_Id" column when Employees table already exists, use the following SQL syntax:

```
ALTER TABLE Employees ADD [CONSTRAINT fKey] FOREIGN KEY (Dept_Id) REFERENCES
Department(Dept_Id);
```

<u>To drop FOREIGN KEY Constraints from a table :</u>

```
ALTER TABLE Employees DROP [FOREIGN KEY Dept_Id | CONSTRAINT fKey];
```

- **CHECK Constraint**

CHECK constraint is used to limit the value range that can be placed in a column. Using check constraint, we can specify conditions for a field, which will be evaluated at the time of entering the data to a column. If the condition evaluates to false, the record violates the constraints and it will not be entered in the table. If we define a CHECK constraint on a single column it allows only certain values for this column. If we define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

```
      CREATE TABLE Employees
       (
       Emp_Id INT(3) PRIMARY KEY,
       Emp_Name VARCHAR(30),
       Salary DOUBLE(8,2) CHECK(Salary > 20000)
      );
```

This query will create a table Employees (Emp_Id,Emp_Name and Salary).CHECK Constraint has been applied on Salary field, thus at the time of entering data into the table, this constraint will be evaluated, if it return false the data will not be entered.

To create a CHECK constraint on the Salary column when Employees table already exists, use the following SQL syntax:

ALTER TABLE Employees ADD [CONSTRAINT CHK_Salary] CHECK (Salary > 20000);

To drop a CHECK constraint, use the following SQL:

ALTER TABLE Employees DROP [CONSTRAINT CHK_Salary | CHECK Salary];

- DEFAULT Constraint

DEFAULT constraint is used to provide a default value for the fields of a table. That is, if at the time of entering new records in the table if the user does not specify any value for these fields then the default value will be assigned to them. The user needs to define these default values.

For example:

```
CREATE TABLE Employees
 (
 Emp_Id INT(3),
 Emp_Name VARCHAR(30),
 Salary DOUBLE(8,2) DEFAULT 20000.00
);
```

This query will create a table **Employees**(Emp_Id, Emp_Name, Salary) with DEFAULT Constraint applied to Salary field, thus **if** no value is entered into Salary field, it will automatically store the **default** value i.e. 20000.00.

To create a DEFAULT constraint on the Salary column when Employees table already exists, use the following SQL syntax:

ALTER TABLE Employees ADD CONSTRAINT CHK_Salary DEFAULT 20000 FOR Salary;

OR

ALTER TABLE Persons ALTER City SET DEFAULT 20000;

To drop a DEFAULT constraint, use the following SQL:

ALTER TABLE Employees ALTER Salary DROP DEFAULT;

- **SQL Arithmetic Function**

A mathematical function executes a mathematical operation usually based on input values that are provided as arguments, and return a numeric value as the result of the operation. Mathematical functions operate on numeric data such as decimal, integer, float, real, smallint, and tinyint. Some Arithmetic functions are –

ABS()	This SQL ABS() returns the absolute value of a number passed as an argument.
CEIL()	This SQL CEIL() will rounded up any positive or negative decimal value within the function upwards.
FLOOR()	The SQL FLOOR() rounded up any positive or negative decimal value down to the next least integer value.

EXP()	The SQL EXP() returns e raised to the n-th power(n is the numeric expression), where e is the base of natural algorithm and the value of e is approximately 2.71828183.
LN()	The SQL LN() function returns the natural logarithm of n, where n is greater than 0 and its base is a number equal to approximately 2.71828183.
MOD()	This SQL MOD() function returns the remainder from a division.
POWER()	This SQL POWER() function returns the value of a number raised to another, where both of the numbers are passed as arguments.
SQRT()	The SQL SQRT() returns the square root of given value in the argument.

SQL Character Function

A character or string function is a function which takes one or more characters or numbers as parameters and returns a character value. Basic string functions offer a number of capabilities and return a string value as a result set.
Some Character functions are –

Functions	**Description**
lower()	The SQL LOWER() function is used to convert all characters of a string to lower case.
upper()	The SQL UPPER() function is used to convert all characters of a string to uppercase.
trim()	The SQL TRIM() removes leading and trailing characters(or both) from a character string.
translate()	The SQL TRANSLATE() function replaces a sequence of characters in a string with another sequence of characters. The function replaces a single character at a time.

Example:

If we want to get the string 'TESTING FOR LOWER FUNCTION' in lower case from the **DUAL** table, the following SQL statement can be used :

```
SELECT LOWER('TESTING FOR LOWER FUNCTION')

AS Testing_Lower

FROM dual;
```

Copy

In the above example, 'Testing_Lower' is a column alias which will come as a column heading to the output.

Output:

```
TESTING_LOWER
--------------------------
testing for lower function
```

Aggregate functions in SQL

As the Basic SQL Tutorial points out, SQL is excellent at aggregating data the way you might in a pivot table in Excel. You will use aggregate functions all the time, so it's important to get comfortable with them. The functions themselves are the same ones you will find in Excel or any other analytics program. We'll cover them individually in the next few lessons. Here's a quick preview:

- COUNT counts how many rows are in a particular column.
- SUM adds together all the values in a particular column.
- MIN return the lowest values in a particular column.
- MAX return the highest values in a particular column.
- AVG calculates the average of a group of selected values.

- **COUNT() with DISTINCT**

1. SELECT COUNT(DISTINCT COMPANY)
2. FROM PRODUCT_MAST;

Output:

```
3
```

Example: COUNT() with GROUP BY

1. SELECT COMPANY, COUNT(*)

2. FROM PRODUCT_MAST
3. GROUP BY COMPANY;

Output:

```
Com1   5
Com2   3
Com3   2
```

Example: COUNT() with HAVING

1. SELECT COMPANY, COUNT(*)
2. FROM PRODUCT_MAST
3. GROUP BY COMPANY
4. HAVING COUNT(*)>2;

Output:

```
Com1   5
Com2   3
```

- **SUM Function**

Sum function is used to calculate the sum of all selected columns. It works on numeric fields only.

Syntax

1. SUM()
2. or
3. SUM([ALL|DISTINCT] expression)

Example: SUM()

1. SELECT SUM(COST)
2. FROM PRODUCT_MAST;

- **AVG function**

The AVG function is used to calculate the average value of the numeric type. AVG function returns the average of all non-Null values.

Syntax

1. AVG()
2. or

3. AVG([ALL|DISTINCT] expression)

Example:

```
SELECT AVG(COST)
FROM PRODUCT_MAST;
```

Output:

```
67.00
```

- **MAX Function**

MAX function is used to find the maximum value of a certain column. This function determines the largest value of all selected values of a column.

Syntax

1. MAX()
2. or
3. MAX([ALL|DISTINCT] expression)

Example:

```
SELECT MAX(RATE)
FROM PRODUCT_MAST;
```

```
30
```

- **MIN Function**

MIN function is used to find the minimum value of a certain column. This function determines the smallest value of all selected values of a column.

Syntax

1. MIN()
2. or
3. MIN([ALL|DISTINCT] expression)

Example:

```
SELECT MIN(RATE)
FROM PRODUCT_MAST;
```

Output:

SQL GROUP BY Statement

The SQL GROUP BY Statement

The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country".

The GROUP BY statement is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.

GROUP BY Syntax

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo	Ana Trujillo	Avda. de la	México	05021	Mexico
3	Antonio Moreno	Antonio	Mataderos	México	05023	Mexico
4	Around the	Thomas	120 Hanover	London	WA1 1DP	UK
5	Berglunds	Christina	Berguvsvägen	Luleå	S-958 22	Sweden

SQL GROUP BY Examples

The following SQL statement lists the number of customers in each country:

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country;
```

The following SQL statement lists the number of customers in each country, sorted high to low:

Example

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
ORDER BY COUNT(CustomerID) DESC;
```

Demo Database

Below is a selection from the "Orders" table in the Northwind sample database:

OrderID	**CustomerID**	**EmployeeID**	**OrderDate**	**ShipperID**
10248	90	5	1996-07-04	3
10249	81	6	1996-07-05	1
10250	34	4	1996-07-08	2

And a selection from the "Shippers" table:

ShipperID	ShipperName
1	Speedy Express
2	United Package
3	Federal Shipping

GROUP BY With JOIN Example

The following SQL statement lists the number of orders sent by each shipper:

Example

```
SELECT Shippers.ShipperName, COUNT(Orders.OrderID) AS NumberOfOrders FROM Order
s
LEFT JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID
GROUP BY ShipperName;
```

The SQL HAVING Clause

The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

HAVING Syntax

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDER BY column_name(s);
```

Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo	Ana Trujillo	Avda. de la	México	05021	Mexico
3	Antonio Moreno	Antonio	Mataderos	México	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover	London	WA1 1DP	UK

5	Berglunds	Christina	Berguvsvägen	Luleå	S-958 22	Sweden

SQL HAVING Examples

The following SQL statement lists the number of customers in each country. Only include countries with more than 5 customers:

Example

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT(CustomerID) > 5;
```

The following SQL statement lists the number of customers in each country, sorted high to low (Only include countries with more than 5 customers):

Example

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT(CustomerID) > 5
ORDER BY COUNT(CustomerID) DESC;
```

Demo Database

Below is a selection from the "Orders" table in the Northwind sample database:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10248	90	5	1996-07-04	3
10249	81	6	1996-07-05	1

10250	34	4	1996-07-08	2

And a selection from the "Employees" table:

EmployeeID	LastName	FirstName	BirthDate	Photo	Notes
1	Davolio	Nancy	1968-12-08	EmpID1.pic	Education includes a BA....
2	Fuller	Andrew	1952-02-19	EmpID2.pic	Andrew received his BTS....
3	Leverling	Janet	1963-08-30	EmpID3.pic	Janet has a BS degree....

More HAVING Examples

The following SQL statement lists the employees that have registered more than 10 orders:

Example

```
SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders
FROM (Orders
INNER JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID)
GROUP BY LastName
HAVING COUNT(Orders.OrderID) > 10;
```

The following SQL statement lists if the employees "Davolio" or "Fuller" have registered more than 25 orders:

Example

```
SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders
FROM Orders
INNER JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID
WHERE LastName = 'Davolio' OR LastName = 'Fuller'
```

```
GROUP BY LastName
HAVING COUNT(Orders.OrderID) > 25;
```

Subquery

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.

Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.

There are a few rules that subqueries must follow −

- Subqueries must be enclosed within parentheses.
- A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
- An ORDER BY command cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY command can be used to perform the same function as the ORDER BY in a subquery.
- Subqueries that return more than one row can only be used with multiple value operators such as the IN operator.
- The SELECT list cannot include any references to values that evaluate to a BLOB, ARRAY, CLOB, or NCLOB.
- A subquery cannot be immediately enclosed in a set function.
- The BETWEEN operator cannot be used with a subquery. However, the BETWEEN operator can be used within the subquery.

Subqueries with the SELECT Statement

Subqueries are most frequently used with the SELECT statement. The basic syntax is as follows −

```
SELECT column_name [, column_name ]
FROM   table1 [, table2 ]
WHERE  column_name OPERATOR
  (SELECT column_name [, column_name ]
  FROM table1 [, table2 ]
  [WHERE])
```

Example

Consider the CUSTOMERS table having the following records −

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  35 | 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        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Now, let us check the following subquery with a SELECT statement.

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

This would produce the following result.

```
+----+----------+-----+---------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY   |
+----+----------+-----+---------+----------+
|  4 | Chaitali |  25 | Mumbai  |  6500.00 |
|  5 | Hardik   |  27 | Bhopal  |  8500.00 |
|  7 | Muffy    |  24 | Indore  | 10000.00 |
+----+----------+-----+---------+----------+
```

Joins.

The **SQL Joins** clause is used to combine records from two or more tables in a database. A **JOIN** is a means for combining fields from two tables by using values common to each.

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

SQL JOIN
A JOIN clause is used to combine rows from two or more tables, based on a related column between them.
Let's look at a selection from the "Orders" table:

OrderID	CustomerID	OrderDate
10308	2	1996-09-18
10309	37	1996-09-19
10310	77	1996-09-20

Then, look at a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Country
1	Alfreds Futterkiste	Maria Anders	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mexico

Notice that the "CustomerID" column in the "Orders" table refers to the "CustomerID" in the "Customers" table. The relationship between the two tables above is the "CustomerID" column.

Then, we can create the following SQL statement (that contains an INNER JOIN), that selects records that have matching values in both tables:

Example

```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate
FROM Orders
INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;
```

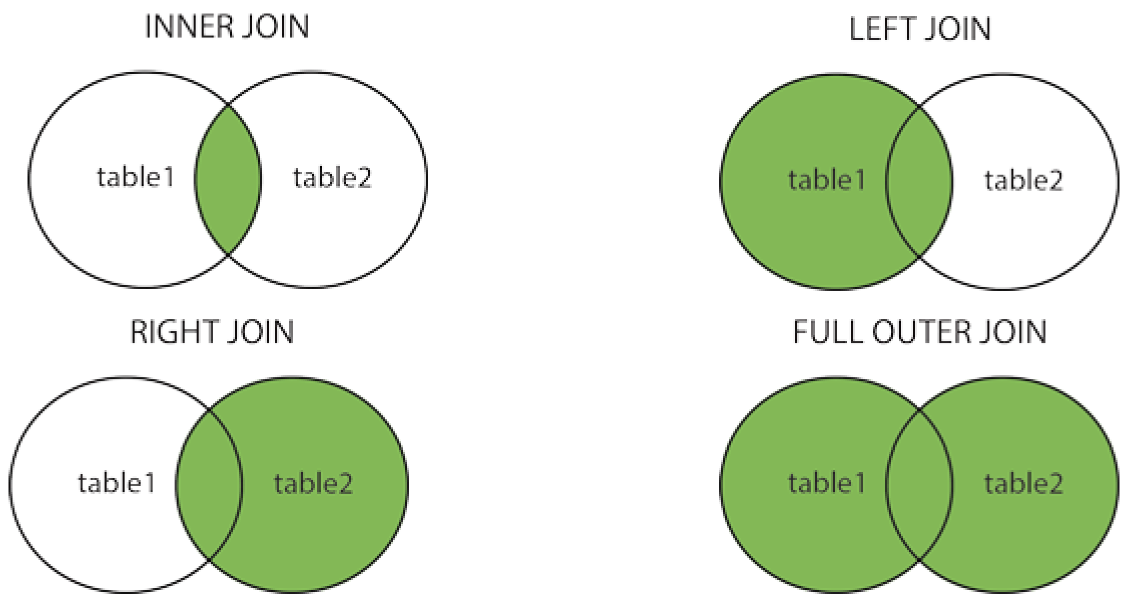
and it will produce something like this:

OrderID	CustomerName	OrderDate
10308	Ana Trujillo Emparedados y helados	9/18/1996
10365	Antonio Moreno Taquería	11/27/1996

10383	Around the Horn	12/16/1996
10355	Around the Horn	11/15/1996
10278	Berglunds snabbköp	8/12/1996

Different Types of SQL JOINs
Here are the different types of the JOINs in SQL:

- **(INNER) JOIN**: Returns records that have matching values in both tables
- **LEFT (OUTER) JOIN**: Returns all records from the left table, and the matched records from the right table
- **RIGHT (OUTER) JOIN**: Returns all records from the right table, and the matched records from the left table
- **FULL (OUTER) JOIN**: Returns all records when there is a match in either left or right table



The SQL **Joins** clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each.

Consider the following two tables –

Table 1 − CUSTOMERS Table

```
+----+----------+-----+-----------+----------+
| 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        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Table 2 − ORDERS Table

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

Now, let us join these two tables in our SELECT statement as shown below.

```
SQL> SELECT ID, NAME, AGE, AMOUNT
  FROM CUSTOMERS, ORDERS
  WHERE  CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result.

```
+----+----------+-----+--------+
| ID | NAME     | AGE | AMOUNT |
+----+----------+-----+--------+
|  3 | kaushik  |  23 |   3000 |
|  3 | kaushik  |  23 |   1500 |
|  2 | Khilan   |  25 |   1560 |
|  4 | Chaitali |  25 |   2060 |
+----+----------+-----+--------+
```

Here, it is noticeable that the join is performed in the WHERE clause. Several operators can be used to join tables, such as =, <, >, <>, <=, >=, !=, BETWEEN, LIKE, and NOT; they can all be used to join tables. However, the most common operator is the equal to symbol.

There are different types of joins available in SQL −

- INNER JOIN − returns rows when there is a match in both tables.
- LEFT JOIN − returns all rows from the left table, even if there are no matches in the right table.
- RIGHT JOIN − returns all rows from the right table, even if there are no matches in the left table.
- FULL JOIN − returns rows when there is a match in one of the tables.SELF JOIN − is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.

The SQL UNION Operator

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

- Each SELECT statement within UNION must have the same number of columns
- The columns must also have similar data types
- The columns in each SELECT statement must also be in the same order

UNION Syntax

SELECT *column_name(s)* FROM *table1*
UNION
SELECT *column_name(s)* FROM *table2*;

- UNION ALL Syntax

The UNION operator selects only distinct values by default. To allow duplicate values, use UNION ALL:

SELECT *column_name(s)* FROM *table1*
UNION ALL
SELECT *column_name(s)* FROM *table2*;

Note: The column names in the result-set are usually equal to the column names in the first SELECT statement in the UNION.

Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico

3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

And a selection from the "Suppliers" table:

SupplierID	SupplierName	ContactName	Address	City	PostalCode	Country
1	Exotic Liquid	Charlotte Cooper	49 Gilbert St.	London	EC1 4SD	UK
2	New Orleans Cajun Delights	Shelley Burke	P.O. Box	New Orleans	70117	USA
3	Grandma Kelly's Homestead	Regina Murphy	707 Oxford Rd.	Ann Arbor	48104	USA

SQL UNION Example

The following SQL statement returns the cities (only distinct values) from both the "Customers" and the "Suppliers" table:

Example

```
SELECT City FROM Customers
UNION
SELECT City FROM Suppliers
ORDER BY City;
```

Note: If some customers or suppliers have the same city, each city will only be listed once, because UNION selects only distinct values. Use UNION ALL to also select duplicate values!

SQL UNION ALL Example

The following SQL statement returns the cities (duplicate values also) from both the "Customers" and the "Suppliers" table:

Example

```
SELECT City FROM Customers
UNION ALL
SELECT City FROM Suppliers
ORDER BY City;
```