

SQL

Objectives:

- **Introduction to SQL**
- **Features of SQL**
- **Queries and Sub-Queries**
- **Set Operations**
- **Relations (Joined , Derived)**
- **Queries under DDL and DML Commands**

- **Embedded SQL**
- **Views**
- **Relational Algebra**
- **Database Modification**
- **QBE and Domain Relational Calculus**

Introduction to SQL :

- SQL(Structured Query Language)
- It is a database computer language designed for the retrieval and management of data in a relational database.
- It is a ANSI (American National Standard Institute) for relational database system.
- All the relational database management system like MySql, MSAccess, Oracle, Sybase and SQL Server use SQL as their standard database language.

RDBMS

- Relational Database Management System is a database management system based on the **relational model** introduced by **E.F. Codd**.
- In a relational model, data is stored in a relation (table) and is represented in form of tuples (rows). It is used to manage relational database. Relational database is a collection of organized tables related to each other and from which data can be accessed easily.

Tables

- It is a collection of data elements organized in terms of rows and columns. Tables can have duplicate row of data but relation can't have duplicate row data.
- Each row in a relation represents a distinct tuple, and it is possible for two or more tuples to have identical values in all attributes (columns). These duplicate tuples are considered distinct records within the relation.

Record or Row or Tuple

- A record or row is called as a row of data is each individual entry that exists in a table.

Attribute

- A table consists of several records(row), each record can be broken down into several smaller parts of data known as Attributes. So it is the properties of relation where its domain is defined to hold certain type of data values.

Null Value

- A Null Value in a table is a value in a field that appears to be blank, which means a field with null value is a field with no value. It is different than zero value.

Non-zero value



null



0



undefined



Relation Schema

- A relation schema describes the structure of the relation, with the name of the relation(name of table), its attributes and their names and type.

Relation Key

- A relation key is an attribute which can uniquely identify a particular tuple(row) in a relation(table).

Features of SQL

- High performance
- High availability
- Scalability and flexibility
- Robust Transactional Support
- High security
- Comprehensive Application_development
- Open source
- Management ease

Queries & Sub-Queries SQL Command

- SQL defines following ways to retrieve and manipulate data stored in an RDBMS.

Statements

- **DDL(Data Definition Language)** : It includes changes to the structure of table like creation of tables, altering tables, deleting tables etc. All DDL commands are auto-committed. That means it saves all the changes permanently in the database.

Command	Description
CREATE	To create new table or database.
ALTER	For alteration
TRUNCATE	Delete data from table
DROP	To drop a table
RENAME	To rename a table

CREATE Command:

- **CREATE** is a DDL SQL command used to create a database or a table in relational database management system.

Creating a Database

- To create a database in RDBMS, **CREATE** command is used.

Syntax

CREATE DATABASE <DB_NAME>;

Example

CREATE DATABASE Test;

- This will create a database named **Test**, which will be an empty schema without any table.

Creating a Table

- CREATE command can also be used to create tables.
- Required to specify the **details of the columns** of the tables too i.e. **name and datatypes** of columns.

Syntax

```
CREATE TABLE <TABLE_NAME>  
(  
    column_name1 datatype1,  
    column_name2 datatype2,  
    column_name3 datatype3,  
    column_name4 datatype4  
);
```

Example

```
CREATE TABLE Student (  
    student_id INT,  
    name VARCHAR(100),  
    age INT  
);
```

ALTER command

- **ALTER** command is used for altering the table structure, such as,
 - to add a column to existing table
 - to rename any existing column
 - to change datatype of any column or to modify its size.
 - to drop a column from the table.

ALTER Command: Add a new Column

- Using ALTER command , we can add a column to any existing table.

- **Syntax**

```
ALTER TABLE table_name  
ADD ( column_name datatype);
```

- **Example**

```
ALTER TABLE student  
ADD (address VARCHAR(200) );
```

- This will add a new column `address` to the table **student**, which will hold data of type varchar which is nothing but string, of length 200.

ALTER Command: Add multiple new Columns

- Using ALTER command , we can even add multiple new columns to any existing table.

- **Syntax**

```
ALTER TABLE table_name  
ADD (  
    column_name1 datatype1,  
    column-name2 datatype2,  
    column-name3 datatype3  
);
```

Example

```
ALTER TABLE student  
ADD(  
    father_name VARCHAR(60),  
    mother_name VARCHAR(60),  
    dob DATE  
);
```

ALTER Command: Modify an existing Column's Datatype:

- ALTER command can also be used to modify data type of any existing column.

- **Syntax**

```
ALTER TABLE table_name  
MODIFY (column_name datatype);
```

- **Example**

```
ALTER TABLE student  
MODIFY ( address varchar(300) );
```

- Remember we added a new column address in the beginning ! The above command will modify the address column of the **student** table, to now hold upto 300 characters.

ALTER Command: Rename a Column

- Using ALTER command you can rename an existing column.

- **Syntax**

```
ALTER TABLE table_name  
RENAME old_column_name TO new_column_name;
```

- **Example**

```
ALTER TABLE student  
RENAME address TO location;
```

- The above command will rename **address** column to **location**.

ALTER Command: Drop a Column

- ALTER command can also be used to drop or remove columns.
- **Syntax**

ALTER TABLE table_name

DROP (column_name); [In sql server, add COLUMN in between DROP and column_name).

- **Example**

```
ALTER TABLE student  
DROP (address);
```

- The above command will drop the address column from the table **student**.

TRUNCATE command

- TRUNCATE command removes all the records from a table. But this command will not destroy the table's structure.
- When we use TRUNCATE command on a table its (auto-increment) primary key is also initialized.

- **Syntax**

TRUNCATE TABLE table_name

- **Example**

TRUNCATE TABLE student;

- The above query will delete all the records from the table **student**.
- In DML commands, we will study about the DELETE command which is also more or less same as the TRUNCATE command.

DROP command

- DROP command completely removes a table from the database. This command will also destroy the table structure and the data stored in it.

- **Syntax**

DROP TABLE table_name

Example

DROP TABLE student;

- The above query will delete the **Student** table completely.

Drop Database

- It can also be used on Databases, to delete the complete database. For example, to drop a database,

- **Example**

`DROP DATABASE Test;`

- The above query will drop the database with name **Test** from the system.

RENAME Query

- RENAME command is used to set a new name for any existing table.

- **Syntax**

RENAME TABLE old_table_name TO new_table_name;

- **Example**

RENAME TABLE student TO students_info;

- The above query will rename the table **student** to **students_info**.

Use Database

- Syntax : Use database_name ;

DML(Data Manipulation Language)

- Data Manipulation Language (DML) statements are used for **managing data** in database.
- DML commands are **not auto-committed**. It means changes made by DML command are not permanent to database, it can be rolled back.

INSERT command

- Insert command is used to insert data into a table.
- **Syntax**

INSERT INTO table_name VALUES (data1, data2, ...) ;

- Example, Consider a table **student** with the following fields.

INSERT INTO student VALUES (101, 'Adam', 15);

s_id	name	age
101	Adam	15

Insert value into only specific columns

- We can use the INSERT command to insert values for only some specific columns of a row.

INSERT INTO student (s_id, name) VALUES (102, 'Alex');

Insert NULL value to a column

- Both the statements below will insert NULL value into **age** column of the **student** table.

INSERT INTO student(s_id, name) values(102, 'Alex');

Or,

INSERT INTO student VALUES (102,'Alex', null);

- The above command will insert only two column values and the other column is set to null.

s_id	name	age
101	Adam	15
102	Alex	NULL

Insert Default value to a column

Suppose, Default for age is 14

```
INSERT INTO student VALUES ( 103 , 'Chris' , default );
```

s_id	name	age
101	Adam	15
102	Alex	NULL
103	Chris	14

SELECT Command

- **SELECT** query is used to retrieve data from a table. (most used SQL query)
- We can retrieve complete table data, or partial by specifying conditions using the **WHERE** clause.

Syntax

```
SELECT  
column_name1, column_name2, column_name3, ... , column_nameN  
FROM table_name;
```

- **Example :** Consider the following **student** table.

SELECT s_id , name , age FROM student;

The above query will fetch information of s_id, name and age columns of the **student** table and display them,

Actual Table

s_id	name	age	gender
101	Sulochan	20	Male
102	Ankush	21	Male
103	Aaradhya	22	Female
104	Roman	23	Male

Fetches Table

s_id	name	age
101	Sulochan	20
102	Ankush	21
103	Aaradhya	22
104	Roman	23

Select all records from a table

- A special character asterisk * is used to address all the column in a query.

Syntax

```
SELECT * FROM student;
```

- Select all the records of **student** table, that means it will show complete dataset of the table.

Select a particular record based on a condition

- **WHERE** clause is used to set a condition,

```
SELECT * FROM student WHERE name = 'Roman';
```

- Result

s_id	name	age	gender
104	Roman	23	Male

Performing Simple Calculations using SELECT Query

- Example

```
SELECT e_id, name, salary+3000 FROM employee;
```

Actual Table

e_id	name	address	salary
201	Roshan	KTM	10000
202	Ritika	DHG	20000
203	Riyan	BHR	30000
204	Aadhi	NPJ	40000
205	Arpita	BKT	10000

Fetch Table

e_id	name	salary+3000
201	Roshan	13000
202	Ritika	23000
203	Riyan	33000
204	Aadhi	43000
205	Arpita	13000

UPDATE Command

- UPDATE command is used to update any record of data in a table.

- **Syntax**

```
UPDATE table_name  
SET  
    column_name = new_value  
WHERE some_condition;
```

Example

```
UPDATE student  
SET  
    age =16  
WHERE s_id =102;
```

- **WHERE** is used to add a condition to any SQL query, we will soon study about it in detail.

Before

s_id	name	age
101	Adam	15
102	Alex	
103	Chris	14

After

s_id	name	age
101	Adam	15
102	Alex	16
103	Chris	14

Question: What if we do not use where clause?

Updating Multiple Columns

UPDATE student

SET

name='Abhi' ,

age=17

where s_id=103;

- The above command will update two columns of the record which has s_id 103.

Before

s_id	name	age
101	Adam	15
102	Alex	16
103	Chris	14

After

s_id	name	age
101	Adam	15
102	Alex	16
103	Abhi	17

DELETE command

- DELETE command is used to delete data from a table.
- **Syntax**

DELETE FROM table_name

Where some_condition;

- **Example**

DELETE FROM student
WHERE s_id=103;

Before

s_id	name	age
101	Adam	15
102	Alex	16
103	Abhi	17

After

s_id	name	age
101	Adam	15
102	Alex	16

Question: What if we do not use where clause?

Isn't DELETE same as TRUNCATE??

- TRUNCATE command is different from DELETE command.
- The delete command will delete all the rows from a table whereas truncate command not only deletes all the records stored in the table, but it also re-initializes the table(like a newly created table).
- **For eg:** If you have a table with 10 rows and an **auto_increment** primary key, and if you use DELETE command to delete all the rows, it will delete all the rows, but will not re-initialize the primary key, hence if you will insert any row after using the DELETE command, the auto_increment primary key will start from 11.
- But in case of TRUNCATE command, primary key is re-initialized, and it will again start from 1

WHERE Clause

- **WHERE** clause is used to specify/apply any condition while retrieving, updating or deleting data from a table.
- This clause is used mostly with SELECT, UPDATE and DELETE query.
- When we specify a condition using the WHERE clause then the query executes only for those records for which the condition specified by the WHERE clause is true.

Syntax

- WHERE clause with a DELETE statement, or any other statement,

```
DELETE FROM table_name  
WHERE [condition];
```

- The WHERE clause is used at the end of any SQL query, to specify a condition for execution.

Example

SELECT statement to display data of the table, based on a condition

- Simple SQL query to display the record for student with s_id as 101

```
SELECT s_id ,name , age FROM student WHERE s_id =101;
```

Before

s_id	name	age
101	Adam	15
102	Alex	16
103	Abhi	17

After

s_id	name	age
101	Adam	15

Applying condition on Text Fields

- Apply the condition on text field.
- In that case we must enclose the value in single quote ' '. Some databases even accept double quotes, but single quotes is accepted by all.

Example

```
SELECT s_id, name, age FROM student WHERE name = 'Adam';
```

Before		
s_id	name	age
101	Adam	15
102	Alex	16
103	Abhi	17

After		
s_id	name	age
101	Adam	15

SQL Data Types

- SQL Data Type specifies the type of data of any object.

Exact Numeric Data Types :

DATA TYPE	FROM	TO
bigint	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
int	-2,147,483,648	2,147,483,647
smallint	-32,768	32,767
tinyint	0	255
bit	0	1
decimal	$-10^{38} + 1$	$10^{38} - 1$
numeric	$-10^{38} + 1$	$10^{38} - 1$
money	-922,337,203,685,477.5808	+922,337,203,685,477.5807
smallmoney	-214,748.3648	+214,748.3647

Date and Time Data Types:

DATA TYPE	DESCRIPTION
datetime	YYYY-MM-DD hh:mm:ss [.nnn]
smalldatetime	YYYY-MM-DD hh:mm:ss
date	Stores a date like June 30, 1991
time	Stores a time of day like 12:30.33 P.M.

Character Strings Data Types

S.No.	DATA TYPE & Description
1	char Maximum length of 8,000 characters.(Fixed length non-Unicode characters)
2	varchar Maximum of 8,000 characters.(Variable-length non-Unicode data).
4	text Variable-length non-Unicode data with a maximum length of 2,147,483,647 characters.

Operator

- An operator is a reserved word or a character used primarily in an **SQL statement's WHERE clause** to perform operation(s), such as comparisons and arithmetic operations.
- Arithmetic Operation can also be used in column selection operations
- These Operators are used to **specify conditions** in an SQL statement and to serve as **conjunctions for multiple conditions** in a statement.
- Some operators are:
 - Arithmetic operators
 - Comparison operators
 - Logical operators

SQL Arithmetic Operators

- Assume '**variable a**' holds 10 and '**variable b**' holds 20, then –

Operator	Example
+ (Addition)	$a + b$ will give 30
- (Subtraction)	$a - b$ will give -10
* (Multiplication)	$a * b$ will give 200
/ (Division)	b / a will give 2
% (Modulus)	$b \% a$ will give 0

SQL Comparison Operators

- Assume '**variable a**' holds 10 and '**variable b**' holds 20, then –

Operator	Description	Example
=	Checks if the values of two operands are equal or not , if yes then condition becomes true.	(a = b) is not true.
!=	Checks if the values of two operands are equal or not , if values are not equal then condition becomes true.	(a != b) is true.
<>	Checks if the values of two operands are equal or not , if values are not equal then condition becomes true.	(a <> b) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.
!<	Checks if the value of left operand is not less than the value of right operand, if yes then condition becomes true.	(a !< b) is false.
!>	Checks if the value of left operand is not greater than the value of right operand, if yes then condition becomes true.	(a !> b) is true.

Example 1:

SELECT * FROM Employee WHERE SALARY > 5000;

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
2	Khaustub	25	Kupondole	1500.00
3	Kaushik	23	Kalimati	2000.00
4	Chetan	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
6	Komal	22	Teku	4500.00
7	Muffy	24	Gaushala	10000.00

ID	NAME	AGE	ADDRESS	SALARY
4	Chetan	25	Kalanki	6500.00
5	Hardik	27	Tripureshw or	8500.00
7	Muffy	24	Gaushala	10000.00

Example 2:

SELECT * FROM Employee WHERE SALARY = 2000;

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
2	Khaustub	25	Kupondole	1500.00
3	Kaushik	23	Kalimati	2000.00
4	Chetan	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
6	Komal	22	Teku	4500.00
7	Muffy	24	Gaushala	10000.00

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
3	Kaushik	23	Kalimati	2000.00

Example 3:

SELECT * FROM Employee WHERE SALARY != 2000;

ID	NAME	AGE	ADDRESS	SALARY	ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00	2	Khaustub	25	Kupondole	1500.00
2	Khaustub	25	Kupondole	1500.00	4	Chetan	25	Kalanki	6500.00
3	Kaushik	23	Kalimati	2000.00	5	Hardik	27	Tripureshwor	8500.00
4	Chetan	25	Kalanki	6500.00	6	Komal	22	Teku	4500.00
5	Hardik	27	Tripureshwor	8500.00	7	Muffy	24	Gaushala	10000.00
6	Komal	22	Teku	4500.00					
7	Muffy	24	Gaushala	10000.00					

Example 4:

SELECT * FROM Employee WHERE SALARY != 2000;

or

SELECT * FROM Employee WHERE SALARY <> 2000;

ID	NAME	AGE	ADDRESS	SALARY	ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00	2	Khaustub	25	Kupondole	1500.00
2	Khaustub	25	Kupondole	1500.00	4	Chetan	25	Kalanki	6500.00
3	Kaushik	23	Kalimati	2000.00	5	Hardik	27	Tripureshwor	8500.00
4	Chetan	25	Kalanki	6500.00	6	Komal	22	Teku	4500.00
5	Hardik	27	Tripureshwor	8500.00	7	Muffy	24	Gaushala	10000.00
6	Komal	22	Teku	4500.00					
7	Muffy	24	Gaushala	10000.00					

Example 5:

SELECT * FROM Employee WHERE SALARY >= 6500;

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
2	Khaustub	25	Kupondole	1500.00
3	Kaushik	23	Kalimati	2000.00
4	Chetan	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
6	Komal	22	Teku	4500.00
7	Muffy	24	Gaushala	10000.00

ID	NAME	AGE	ADDRESS	SALARY
4	Chetan	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
7	Muffy	24	Gaushala	10000.00

SQL Logical Operators

S.No.	Operator & Description
1	ALL The ALL operator is used to compare a value to all values in another value set. True if all of a set of comparison are TRUE. <code>10 > all(2, 4, 5) -> True</code> <code>10 > all(12, 4, 6) -> False</code>
2	ANY The ANY operator is used to compare a value to any applicable value in the list as per the condition. True is any one of a set of comparison is TRUE. <code>10 > any(2, 4, 5) -> True</code> <code>10 > any(12, 4, 6) -> True</code> <code>10 > any(12, 14, 16) -> False</code>
3	AND The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause. AND makes it necessary for all conditions to be true.
4	OR The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause. OR makes it necessary for one condition to be true of multiple conditions.

5	BETWEEN The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value. Eg. age between 16 and 60
6	EXISTS The EXISTS operator is used to search for the presence of a row in a specified table that meets a certain criterion. The EXISTS operator returns TRUE if the subquery returns one or more records.
7	IN The IN operator is used to compare a value to a list of literal values that have been specified. Eg: age in (16, 18, 24)
8	LIKE The LIKE operator is used to compare a value to similar values in a table in the database. It is used with wildcard character (% =percentage symbol , _ = underscore character).
9	NOT The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. This is a negate operator.
10	IS NULL The NULL operator is used to compare a value with a NULL value.

Example 1:

SELECT * FROM Employee WHERE AGE >= 25 AND SALARY >= 6500;

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
2	Khaustub	25	Kupondole	1500.00
3	Kaushik	23	Kalimati	2000.00
4	Chetan	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
6	Komal	22	Teku	4500.00
7	Muffy	24	Gaushala	10000.00

ID	NAME	AGE	ADDRESS	SALARY
4	Chetan	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00

Example 2:

SELECT * FROM Employee WHERE AGE >= 25 OR SALARY >= 6500;

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
2	Khaustub	25	Kupondole	1500.00
3	Kaushik	23	Kalimati	2000.00
4	Chetan	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
6	Komal	22	Teku	4500.00
7	Muffy	24	Gaushala	10000.00

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
2	Khaustub	25	Kupondole	1500.00
4	Chetan	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
7	Muffy	24	Gaushala	10000.00

Example 3:

SELECT * FROM Employee WHERE AGE IS NOT NULL;

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
2	Khaustub	25	Kupondole	1500.00
3	Kaushik	23	Kalimati	2000.00
4	Chetan	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
6	Komal	22	Teku	4500.00
7	Muffy	24	Gaushala	10000.00

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
2	Khaustub	25	Kupondole	1500.00
3	Kaushik	23	Kalimati	2000.00
4	Chetan	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
6	Komal	22	Teku	4500.00
7	Muffy	24	Gaushala	10000.00

Example 4:

SELECT * FROM Employee WHERE NAME LIKE 'Ko%';

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
2	Khaustub	25	Kupondole	1500.00
3	Kaushik	23	Kalimati	2000.00
4	Chetan	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
6	Komal	22	Teku	4500.00
7	Muffy	24	Gaushala	10000.00

ID	NAME	AGE	ADDRESS	SALARY
6	Komal	22	Teku	4500.00

Example 5:

SELECT * FROM Employee WHERE AGE IN (25, 27);

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
2	Khaustub	25	Kupondole	1500.00
3	Kaushik	23	Kalimati	2000.00
4	Chetan	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
6	Komal	22	Teku	4500.00
7	Muffy	24	Gaushala	10000.00

ID	NAME	AGE	ADDRESS	SALARY
2	Khaustub	25	Kupondole	1500.00
4	Chetan	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00

Example 6:

SELECT * FROM Employee WHERE AGE BETWEEN 25 AND 27;

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
2	Khaustub	25	Kupondole	1500.00
3	Kaushik	23	Kalimati	2000.00
4	Chetan	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
6	Komal	22	Teku	4500.00
7	Muffy	24	Gaushala	10000.00

ID	NAME	AGE	ADDRESS	SALARY
2	Khaustub	25	Kupondole	1500.00
4	Chetan	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00

Example 7:

SELECT AGE FROM Employee WHERE EXISTS (SELECT AGE FROM Employee WHERE SALARY > 6500);

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
2	Khaustub	25	Kupondole	1500.00
3	Kaushik	23	Kalimati	2000.00
4	Chetan	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
6	Komal	22	Teku	4500.00
7	Muffy	24	Gaushala	10000.00

AGE
32
25
23
25
27
22
24

Syntax for ALL operator:

```
SELECT column_name(s)  
FROM table_name
```

```
WHERE column_name operator ALL  
(SELECT column_name  
FROM table_name  
WHERE condition);
```

Syntax for ANY operator:

```
SELECT column_name(s)  
FROM table_name
```

```
WHERE column_name operator ANY  
(SELECT column_name  
FROM table_name  
WHERE condition);
```

Example 8:

SELECT * FROM Employee WHERE AGE > ALL (SELECT AGE FROM Employee WHERE SALARY > 6500);

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
2	Khaustub	25	Kupondole	1500.00
3	Kaushik	23	Kalimati	2000.00
4	Chetan	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
6	Komal	22	Teku	4500.00
7	Muffy	24	Gaushala	10000.00

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00

Example 9:

SELECT * FROM Employee WHERE AGE > ANY (SELECT AGE FROM Employee WHERE SALARY > 6500);

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
2	Khaustub	25	Kupondole	1500.00
3	Kaushik	23	Kalimati	2000.00
4	Chetan	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
6	Komal	22	Teku	4500.00
7	Muffy	24	Gaushala	10000.00

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
2	Khaustub	25	Kupondole	1500.00
4	Chetan	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00

SQL CONSTRAINTS

- SQL Constraints are rules used to **limit the type of data** that can go into a table, to maintain the **accuracy and integrity** of the data inside table.
- **Constraints can be divided into the following two types:**
 - Column level constraints:** Limits only column data.
 - Table level constraints:** Limits whole table data.
- **Most used constraints**
 - NOT NULL
 - DEFAULT
 - UNIQUE
 - PRIMARY KEY
 - FOREIGN KEY
 - CHECK

NOT NULL Constraint

- **NOT NULL** constraint restricts a column from having a NULL value.
- Once **NOT NULL** constraint is applied to a column, you **cannot pass a null value** to that column.
- It enforces a column to contain a **proper value**.
- One important point to note about this constraint is that it **cannot be defined at table level**.

Syntax

```
CREATE TABLE table_name  
(  
    Column_name1 datatype NOT NULL,  
    Column_name2 datatype ,  
    ...  
);
```

Example

```
CREATE TABLE CUSTOMER(  
    ID INT NOT NULL,  
    NAME VARCHAR (20) NOT NULL,  
    AGE INT NOT NULL,  
    salary DECIMAL (18, 2)  
);
```

- If CUSTOMER table has already been created, then to add a NOT NULL constraint to the SALARY column in Oracle and MySQL, you would write a query like the one that is shown in the following code block.

```
ALTER TABLE CUSTOMER  
MODIFY SALARY DECIMAL (18, 2) NOT NULL;
```

Syntax(In SQL SERVER)

```
ALTER TABLE table_name  
ALTER COLUMN Column_name datatype NOT NULL;
```

To remove Constraint (In SQL SERVER,)

```
ALTER TABLE table_name  
ALTER COLUMN Column_name datatype NULL;
```

MODIFY in MySQL and **ALTER** in SQL is used to modify the data type of columns.

DEFAULT Constraints

- The DEFAULT constraint provides a default value to a column when the INSERT INTO statement does not provide a specific value.

Syntax: (In SQL SERVER)

```
CREATE TABLE table_name  
(  
    Column_name1 datatype,  
    Column_name2 datatype DEFAULT default_value,  
    ...  
);
```

Example

```
CREATE TABLE CUSTOMER  
(  
    ID INT NOT NULL ,  
    NAME VARCHAR (20) NOT NULL,  
    AGE INT NOT NULL,  
    SALARY DECIMAL (18, 2) DEFAULT 5000.00,  
);
```

- If the CUSTOMER table has already been created, then to add a DEFAULT constraint to the SALARY column, you would write a query like the one which is shown in the code block below.

```
ALTER TABLE CUSTOMER  
MODIFY SALARY DECIMAL (18, 2) DEFAULT 5000.00;
```

Syntax (In SQL SERVER),

```
ALTER TABLE table_name  
ADD CONSTRAINT Constraint_name DEFAULT default _value FOR Column_name;
```

Drop Default Constraint

- To drop a DEFAULT constraint, use the following SQL query.

Syntax:

```
ALTER TABLE table_name  
ALTER COLUMN column_name DROP DEFAULT;
```

Example:

```
ALTER TABLE CUSTOMER  
ALTER COLUMN SALARY DROP DEFAULT;
```

UNIQUE Constraint

- **UNIQUE** constraint ensures that a field or column will only have unique values. A **UNIQUE** constraint field will not have duplicate data.
- This constraint can be applied at **column level** or **table level**.

```
CREATE TABLE CUSTOMER
```

```
(
```

```
    ID INT NOT NULL,
```

```
    NAME VARCHAR (20) NOT NULL,
```

```
    AGE INT NOT NULL UNIQUE,
```

```
    SALARY DECIMAL (18, 2)
```

```
);
```

Syntax: (IN SQL SERVER)

```
CREATE TABLE table_name(  
    Column_name1 datatype UNIQUE,  
    Column_name2 datatype ,  
    ...  
);
```

OR,

```
CREATE TABLE table_name  
(  
    Column_name1 datatype ,  
    Column_name2 datatype ,.....,  
    CONSTRAINT Constraint_name UNIQUE (column_name1,column_name2..)  
);
```

- If the CUSTOMER table has already been created, then to add a **UNIQUE** constraint to the AGE column. You would write a statement like the query that is given in the code block below.

```
ALTER TABLE CUSTOMER  
MODIFY AGE INT NOT NULL UNIQUE;
```

- In SQL SERVER,

```
ALTER TABLE table_name  
ADD CONSTRAINT Constraint_name UNIQUE (column_name1,column_name2...);
```

- Example

```
ALTER TABLE CUSTOMER  
ADD CONSTRAINT myUniqueConstraint UNIQUE (AGE, SALARY);
```

DROP a UNIQUE Constraint

- To drop a UNIQUE constraint, use the following SQL query.

In SQL SERVER:

```
ALTER TABLE table_name  
DROP CONSTRAINT Constraint_name;
```

Example:

```
ALTER TABLE CUSTOMER  
DROP CONSTRAINT myUniqueConstraint;
```

MySQL Server:

```
ALTER TABLE CUSTOMER  
DROP INDEX myUniqueConstraint;
```

Primary Key Constraint

- Primary key constraint **uniquely identifies each record** in a database. A Primary Key must contain **unique value** and it must **not contain null value**.
- Usually Primary Key is used to **index the data** inside the table.

Example to Create Primary Key

```
CREATE TABLE CUSTOMER (  
    ID INT NOT NULL,  
    NAME VARCHAR (20) NOT NULL,  
    AGE INT NOT NULL,  
    ADDRESS CHAR (25) ,  
    SALARY DECIMAL (18, 2),  
    CONSTRAINT pk_id PRIMARY KEY (ID)  
);
```

Syntax: (IN SQL SERVER)

```
CREATE TABLE table_name
```

```
(  
    Column_name1 datatype PRIMARY KEY,  
    Column_name2 datatype ,  
    ...  
);
```

OR,

```
CREATE TABLE table_name
```

```
(  
    Column_name1 datatype,  
    Column_name2 datatype ,  
    ...,  
    CONSTRAINT Constraint_name PRIMARY KEY (column_name1, ...)  
);
```


- To create a PRIMARY KEY constraint on the "ID" and "NAMES" columns when CUSTOMER table already exists, use the following SQL syntax.

(In SQL SERVER):

```
ALTER TABLE table_name  
ADD CONSTRAINT Constraint_name PRIMARY KEY (column_name1,column_name2...);
```

Example

```
ALTER TABLE CUSTOMER  
ADD CONSTRAINT PK_CUSTID PRIMARY KEY (ID, NAME);
```

NOTE – If you use the ALTER TABLE statement to add a primary key, the primary key column(s) should have already been declared to not contain NULL values (when the table was first created).

To DROP Primary Key

You can clear the primary key constraints from the table with the syntax given below.

```
ALTER TABLE CUSTOMER  
DROP PRIMARY KEY ;
```

(In SQL SERVER)

```
ALTER TABLE table_name  
DROP CONSTRAINT Constraint_name;
```

Foreign Key Constraint

- FOREIGN KEY is used to relate two tables.
- FOREIGN KEY constraint is also used to **restrict actions that would destroy links between tables.**

Customer_Detail Table

c_id	Customer_name	address
101	Royal	NPJ
102	Sushil	BKT
103	Sanjog	BHR

Order_Detail Table

Order_id	Order_name	c_id
10	O1	101
11	O2	103
12	O3	102

- In **Customer_Detail** Table , **c_id** is the primary key which is set as foreign key in **Order_Detail** table. The value that is entered in **c_id** which is set as foreign key in **Order_Detail** table must be present in **Customer_Detail** table where it is set as primary key. This prevents invalid data to be inserted into **c_id** column of **Order_Detail** table.
- If you try to insert any incorrect data, DBMS will return error and will not allow you to insert the data.

Syntax: (IN SQL SERVER)

```
CREATE TABLE table_name
(
    Column_name1 datatype,
    ...,
    Column_name2 datatype FOREIGN KEY REFERENCES parent_table_name (column_name)
);
```

OR,

```
CREATE TABLE table_name
(
    Column_name1 datatype,
    Column_name2 datatype ,
    ...,
    CONSTRAINT Constraint_name FOREIGN KEY (column_name1,column_name2..) REFERENCES parent_table_name
    (column_name1,column_name2...)
);
```

- **Using FOREIGN KEY constraint at Table Level**

```
CREATE TABLE Order_Detail  
(  
    Order_id int PRIMARY KEY,  
    Order_name varchar(60) NOT NULL,  
    c_id int FOREIGN KEY REFERENCES Customer_Detail(c_id)  
);
```

Using ALTER Command:

```
ALTER TABLE table_name  
ADD FOREIGN KEY (column_name1,column_name2..) REFERENCES parent_table_name  
(column_name1,column_name2...);
```

Example:

```
ALTER TABLE Order_Detail  
ADD FOREIGN KEY (c_id) REFERENCES Customer_Detail(c_id);
```

Behaviour of Foreign Key Column on Delete

- When two tables are connected with Foreign key, and certain data in the main table is deleted, for which a record exists in the child table, then we must have some mechanism to save the integrity of data in the child table.

On Delete Cascade :

This will remove the record from child table, if that value of foreign key is deleted from the main table.

On Delete SET Null :

This will set all the values in that record of child table as NULL, for which the value of foreign key is deleted from the main table.

- If we don't use any of the above, then we cannot delete data from the main table for which data in child table exists. We will get an error if we try to do so : **ERROR : record in child table exist.**

To DROP FOREIGN KEY Constraint,

```
ALTER TABLE table_name  
DROP CONSTRAINT Constraint_name;
```

CHECK Constraint

- **CHECK** constraint is used to restrict the value of a column between a range.
- It performs check on the values, before storing them into the database.
- Example:
CHECK with AGE column, so that you cannot have any CUSTOMER who is below 18 years.

```
CREATE TABLE CUSTOMER(  
    ID INT NOT NULL,  
    NAME VARCHAR (20) NOT NULL,  
    AGE INT NOT NULL CHECK (AGE >= 18),  
    ADDRESS CHAR (25) ,  
    SALARY DECIMAL (18, 2),  
    PRIMARY KEY (ID)  
);
```

- Syntax: (In SQL SERVER)

```
CREATE TABLE table_name  
(  
    Column_name1 datatype,  
    Column_name2 datatype CHECK (Condition),  
    ...  
);
```

OR,

```
CREATE TABLE table_name  
(  
    Column_name1 datatype,  
    Column_name2 datatype ,.....,  
    CONSTRAINT Constraint_name CHECK (Condition)  
);
```

- If the CUSTOMER table has already been created, then to add a CHECK constraint to AGE column, you would write a statement like the one given below.

```
ALTER TABLE CUSTOMER  
MODIFY AGE INT NOT NULL CHECK (AGE >= 18 );
```

In SQL SERVER

```
ALTER TABLE table_name  
ADD CONSTRAINT Constraint_name CHECK(Condition);
```

Example

```
ALTER TABLE CUSTOMER  
ADD CONSTRAINT myCheckConstraint CHECK(AGE >= 18);
```


DROP a CHECK Constraint

- To drop a CHECK constraint, use the following SQL syntax. This syntax does not work with MySQL.

```
ALTER TABLE table_name  
DROP CONSTRAINT Constraint_name;
```

Example

```
ALTER TABLE CUSTOMER  
DROP CONSTRAINT myCheckConstraint;
```

INDEX Constraints

- The INDEX is used **to create and retrieve data** from the database **very quickly**.
- An Index can be created by using a **single or group of columns** in a table.
- Proper indexes are good for performance in large databases, but you need to be careful while creating an index.
- **Syntax**

```
CREATE INDEX index_name  
ON table_name ( column1, column2.....);
```
- To create an INDEX on the AGE column, to optimize the search on customers for a specific age, you can use the following SQL syntax which is given below –

```
CREATE INDEX idx_age  
ON CUSTOMER ( AGE );
```

DROP an INDEX Constraint

- To drop an INDEX constraint, use the following MySQL syntax.

```
ALTER TABLE CUSTOMER  
DROP INDEX idx_age;
```

[In SQL SERVER,]

```
DROP INDEX table_name.Index_name;
```

Order By Clause

- It is used to sort data in ascending or descending order based on one or more columns.

Syntax:

```
Select column_list  
From table_name  
[Where condition]  
[order by column1, column2, .... , columnN ] [ASC|DESC];
```

Example :

```
Select * from customer  
Order by Name, Salary;
```

```
Select * from customer  
Order by Name DESC;
```

Limit Clause

- It is used to specify the number of records to return.

Syntax:

```
Select column_list  
From table_name  
[Where condition]  
[order by column1, column2, .... , columnN ] [ASC|DESC]  
LIMIT offset, number;
```

Example :

```
Select * from customer  
Order by Name, Salary  
limit 20, 10;
```

```
Select * from customer  
Order by Name DESC  
limit 20;
```

Group By Clause

- It is used in collaboration with the SELECT statement to arrange identical data into groups.
- It follows WHERE clause in SELECT statements and precedes the order by clause.

Syntax :	
	Select column1, column2 From table-name [Where condition] [Group by column1, column2] [Order by column1, column2]
Example :	Select Name, Sum(Salary) From customer Group By Name

HAVING Clause

- It filters records that work on summarized **GROUP BY** results.
- It applies to summarized group records, whereas **WHERE** applies to individual records.
- Only the groups that meet the **HAVING** criteria will be returned.
- **HAVING** requires that a **GROUP BY** clause is present. **WHERE** and **HAVING** can be in the same query.
- **Syntax :**

Select column-names
From table-name
Where condition
Group By column-names
Having condition
Order By column-names
Limit offset, number;

Example :

Select COUNT(ID), Country
From customer
Group By Country
Having COUNT(ID) > 10

DISTINCT Keyword

- It used in conjunction with the SELECT statements to eliminate all the duplicate records and fetching only unique records.
- Multiple duplicate records in table will be eliminated.

- **Syntax :**

```
SELECT DISTINCT column1, ....., columnN  
From table-name [Where condition]
```

- **Example :**

```
SELECT DISTINCT Salary  
From customer Order By Salary;
```


TOP Clause

- It is used to fetch a TOP N number or X percent records from the table.
- [Note : MySQL does not support TOP clause, MySQL supports LIMIT clause and Oracle supports Row_num command]
- **Syntax :**
 Select Top number/percent column-name
 From table-name
 [Where condition];
- **Example :**
 - Select TOP 3 * from Customer; [For sql server]
 - Select * from customer limit 3; [For MySQL]
 - Select * from customer Where ROWNUM <=3; [For oracle]

SQL Function:

- SQL provides many built-in functions to perform operations on data.
- These functions are useful while performing mathematical calculations, string concatenations, sub-strings etc.
- Five Different Categories of Function
 1. Number Function
 2. **Aggregate Function**
 3. Character Function
 4. Conversion Function
 5. Date Function

Aggregate Function

- It is used to **accumulate information** from **multiple tuples** forming a **single tuple** summary.

AVG() Function

- Average returns average value after calculating it from values in a numeric column.

Syntax

```
SELECT AVG(column_name) FROM table_name
```

Example

```
SELECT AVG(salary) FROM Emp;
```

e_id	name	age	salary
401	Aayush	22	9000
402	Alish	29	8000
403	Aashu	34	6000
404	Milan	44	10000
405	Sima	35	8000

AVG (salary)
8200

COUNT() Function

- Count returns the number of rows present in the table either based on some condition or without condition.

Syntax

```
SELECT COUNT(column_name) FROM table-name
```

Example:

```
SELECT COUNT(name) FROM Emp WHERE salary = 8000;
```

e_id	name	age	salary
401	Aayush	22	9000
402	Alish	29	8000
403	Aashu	34	6000
404	Milan	44	10000
405	Sima	35	8000

Count(name)
2

- **Example of COUNT (DISTINCT)**

SELECT COUNT (DISTINCT salary) FROM Emp;

e_id	name	age	salary
401	Aayush	22	9000
402	Alish	29	8000
403	Aashu	34	6000
404	Milan	44	10000
405	Sima	35	8000

COUNT (DISTINCT salary)
4

FIRST() Function

- FIRST function returns first value of a selected column.
- **Syntax**

```
SELECT FIRST(column_name) FROM table-name;
```

- **Example**

```
SELECT FIRST(salary) FROM Emp;
```

e_id	name	age	salary
401	Aayush	22	9000
402	Alish	29	8000
403	Aashu	34	6000
404	Milan	44	10000
405	Sima	35	8000

FIRST(salary)
9000

LAST() Function

- LAST() Function returns the last value of the selected column.

- **Syntax**

```
SELECT LAST(column_name) FROM table-name;
```

- **Example**

```
SELECT LAST (salary) FROM Emp;
```

e_id	name	age	salary
401	Aayush	22	9000
402	Alish	29	8000
403	Aashu	34	6000
404	Milan	44	10000
405	Sima	35	8000

LAST(salary)
8000

MAX() Function

- Max() Function returns maximum value from selected column of the table.
- **Syntax**

```
SELECT MAX(column_name) from table-name;
```

Example

```
SELECT MAX (salary) FROM Emp;
```

e_id	name	age	salary
401	Aayush	22	9000
402	Alish	29	8000
403	Aashu	34	6000
404	Milan	44	10000
405	Sima	35	8000

MAX(salary)
10000

MIN() Function

- MIN function returns minimum value from a selected column of the table.

- **Syntax**

SELECT MIN(column_name) from table-name;

- **Example**

SELECT MIN(salary) FROM Emp;

e_id	name	age	salary
401	Aayush	22	9000
402	Alish	29	8000
403	Aashu	34	6000
404	Milan	44	10000
405	Sima	35	8000

MIN(salary)
6000

SUM() Function

- SUM function returns total sum of a selected columns numeric values.

- **Syntax**

```
SELECT SUM(column_name) from table-name;
```

- **Example**

```
SELECT SUM(salary) FROM Emp;
```

e_id	name	age	salary
401	Aayush	22	9000
402	Alish	29	8000
403	Aashu	34	6000
404	Milan	44	10000
405	Sima	35	8000

SUM(salary)
41000

Sub-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.
- 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 :

1. Subqueries must be enclosed within parentheses.
2. A subquery can have only one column in the **SELECT** clause
3. An **ORDER BY** command cannot be used in a subquery, although the main query can use an **ORDER BY**.
4. The **GROUP BY** command can be used to perform the same function as the **ORDER BY** in a subquery.
5. Subqueries that return more than one row can only be used with multiple value operators such as the **IN, ALL, ANY, EXIST** operator.
6. 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.

- **Syntax**

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

Example

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

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
2	Khaustub	25	Kupondole	1500.00
3	Kaushik	23	Kalimati	2000.00
4	Chetan	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
6	Komal	22	Teku	4500.00
7	Muffy	24	Gaushala	10000.00

ID	NAME	AGE	ADDRESS	SALARY
4	Chetan	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
7	Muffy	24	Gaushala	10000.00

Subqueries with the INSERT Statement

- Subqueries also can be used with INSERT statements. The INSERT statement uses the data returned from the subquery to insert into another table. The selected data in the subquery can be modified with any of the character, date or number functions.
- **Syntax**

```
INSERT INTO table_name  
[ (column1 [, column2 ]) ]  
(SELECT [ *|column1 [, column2 ]  
FROM table1 [, table2 ]  
[ WHERE VALUE OPERATOR ] )
```

Example

- Consider a table Employee_BKP with similar structure as Employee table. Now to copy the complete Employee table into the Employee _BKP table, you can use the following syntax.

```
INSERT INTO Employee _BKP  
SELECT * FROM Employee;
```

What is result?

```
INSERT INTO employee_BKP  
(employee_id, employee_name, age)  
VALUES  
(1, 'John Smith', (SELECT AVG(age) FROM employees));
```

Subqueries with the UPDATE Statement

- The subquery can be used in conjunction with the UPDATE statement. Either single or multiple columns in a table can be updated when using a subquery with the UPDATE statement.
- **Syntax**

```
UPDATE TABLE_NAME  
SET  
    column_name = new_value  
[ WHERE COLUMN OPERATOR  
(SELECT COLUMN_NAME  
FROM TABLE_NAME  
WHERE cond) ]
```


Example

- Assuming, we have Employee_BKP table available which is backup of Employee table. The following example updates SALARY by 0.25 times in the Employee table for all the customers whose AGE is greater than or equal to 27.

```
UPDATE Employee
SET SALARY = SALARY * 0.25
WHERE AGE IN (SELECT AGE FROM Employee_BKP
WHERE AGE >= 27 );
```

Subqueries with the DELETE Statement

- The subquery can be used in conjunction with the DELETE statement like with any other statements mentioned above.
- Syntax:

```
DELETE FROM TABLE_NAME  
[ WHERE column OPERATOR  
(SELECT COLUMN_NAME  
FROM TABLE_NAME)  
(WHERE) ]
```

Example

- Assuming, we have a Employee_BKP table available which is a backup of the Employee table. The following example deletes the records from the Employee table for all the customers whose AGE is greater than or equal to 27.

```
DELETE FROM Employee
WHERE AGE IN
( SELECT AGE FROM
Employee_BKP WHERE AGE >= 27 );
```

Like Clause

- The SQL Server LIKE is a logical operator that determines **if a character string matches a specified pattern**.
- A pattern may include **regular characters** and **wildcard characters**.
- The LIKE operator is used in the **WHERE clause** of the SELECT, UPDATE and DELETE statements to filter rows based on pattern matching.
- The LIKE operator returns TRUE if the column or expression matches the specified pattern. To negate the result of the LIKE operator, you use the NOT operator.

Pattern

- The pattern is a sequence of characters to search for in the column or expression. It can include the following valid wildcard characters:
 - The percent wildcard (%) : any string of zero or more characters.
 - The underscore (_) wildcard : any single character.
 - The [list of characters] wildcard : any single character within the specified set.
 - The [character-character] : any single character within the specified range.
 - The [^] : any single character not within a list or a range.

Escape character

- The **escape character** instructs the LIKE operator **to treat the wildcard characters as the regular characters**. The escape character has no default value and must be evaluated to only one character.
- column LIKE pattern [ESCAPE escape_character]
- By default ‘\’ is escape_character

Question	Solution
find customers whose last name starts with the letter 'z'	
find customers whose last name ends with the string er	
find customers whose last name starts with the letter t and ends with the letter s	
find customers where the second character is the letter u	
find customers where the first character in the last name is Y or Z	
finds customers where the first character in the last name is the letter in the range A through C	
find customers where the first character in the last name is not the letter in the range A through X	
find customers where the first character in the first name is not the letter A	

Question	Solution
find customers whose last name starts with the letter 'z'	Select * FROM customers Where last_name LIKE 'z%' ;
find customers whose last name ends with the string er	SELECT * FROM customers WHERE last_name LIKE '%er' ;
find customers whose last name starts with the letter t and ends with the letter s	SELECT * FROM customers WHERE last_name LIKE 't % s';
find customers where the second character is the letter u	SELECT * FROM customers WHERE last_name LIKE '_u%' ;
find customers where the first character in the last name is Y or Z	SELECT * FROM customers WHERE last_name LIKE '[YZ]%'
finds customers where the first character in the last name is the letter in the range A through C	SELECT * FROM customers WHERE last_name LIKE '[A-C]%'
find customers where the first character in the last name is not the letter in the range A through X	SELECT * FROM customers WHERE last_name LIKE '[^A-X]%'
find customers where the first character in the first name is not the letter A	SELECT c* FROM customers WHERE first_name NOT LIKE 'A%'

SET Operations

- SQL supports few Set operations which can be performed on the table data.
- Types of SET operations:
 1. UNION
 2. UNION ALL
 3. INTERSECT
 4. MINUS

UNION Operation

- **UNION** is used to combine the results of two or more **SELECT** statements.
- However, it will **eliminate duplicate rows** from its result set.
- In case of union, **number of columns and data type** must be same in both the tables, on which UNION operation is being applied.

Example of UNION

First Table

s_id	name
1	Gopal
2	Pritam
3	Sumona

Second Table

s_id	name
3	Sumona
4	Aarav
5	Suman

- Union SQL query will be,
SELECT * FROM First
UNION
SELECT * FROM Second

Result will be like this,

s_id	name
1	Gopal
2	Pritam
3	Sumona
4	Aarav
5	Suman

UNION ALL

This operator is similar to UNION. But it also shows the duplicate rows.

Example:

```
SELECT * FROM First  
UNION ALL  
SELECT * FROM Second;
```

s_id	name
1	Gopal
2	Pritam
3	Sumona
3	Sumona
4	Aarav
5	Suman

INTERSECT

- Intersect operator is used to combine two SELECT statements, but it only returns the records which are common from both SELECT statements.
- The number of columns and data type must be same.

[NOTE: MySQL does not support INTERSECT operator.]

Example:

```
SELECT * FROM First  
INTERSECT  
SELECT * FROM Second;
```

s_id	name
3	Sumona

MINUS

- The Minus operation combines results of two SELECT statements and return only those which belongs to the first set of the result.

Example:

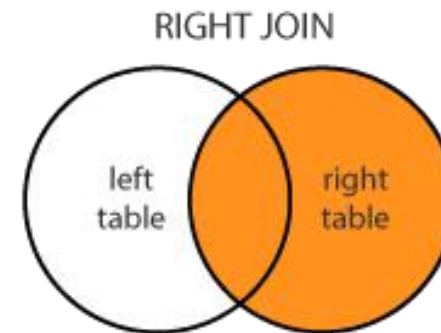
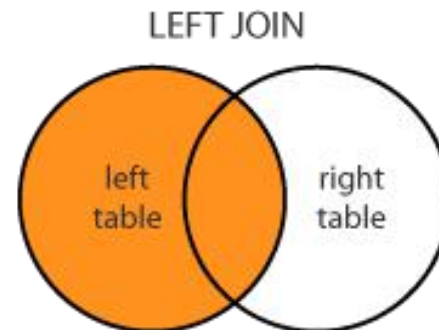
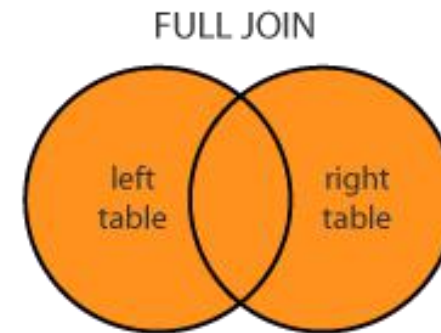
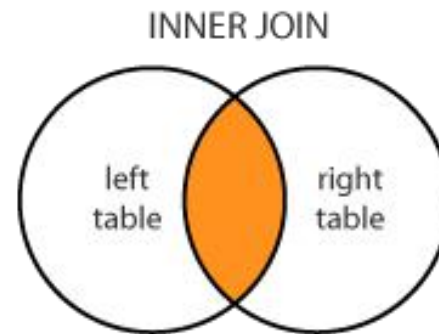
```
SELECT * FROM First  
MINUS  
SELECT * FROM Second;
```

s_id	name
1	Gopal
2	Pritam

Relations (Joined & Derived)

SQL join

- A SQL join is an instruction to combine data from two sets of data (i.e. two tables).
- Types of SQL Joins:
 - inner,
 - left,
 - right, and
 - full



Cross JOIN or Cartesian Product

- This JOIN returns a table which consists of records which **combines each row from the first table with each row of the second table.**

- **Syntax**

```
SELECT column-name-list  
FROM table-name1  
CROSS JOIN table-name2;
```

- **Example**

```
SELECT * FROM class  
CROSS JOIN class_info
```

class		class_info	
ID	Name	ID	Address
1	Abhi	1	DHG
2	Alish	2	BHR
4	Aayush	3	KTM

Result

ID	Name	ID	Address
1	Abhi	1	DHG
2	Alish	1	DHG
4	Aayush	1	DHG
1	Abhi	2	BHR
2	Alish	2	BHR
4	Aayush	2	BHR
1	Abhi	3	KTM
2	Alish	3	KTM
4	Aayush	3	KTM

INNER JOIN

Inner join results is based on matched data as per the equality condition specified in the SQL query.

- **Syntax**

```
SELECT column-name-list  
FROM table-name1  
INNER JOIN table-name2  
ON table-name1.column-name = table-name2.column-name;
```

- **Example**

```
SELECT *  
FROM class  
INNER JOIN class_info  
ON class.ID = class_info.ID;
```

class		class_info	
ID	Name	ID	Address
1	Abhi	1	DHG
2	Alish	2	BHR
4	Aayush	3	KTM

Result

ID	Name	ID	Address
1	Abhi	1	DHG
2	Alish	2	BHR

NATURAL JOIN

- Natural Join is a type of Inner join which is based **on column having same name and same datatype** present in **both the tables** to be joined.

- **Syntax**

```
SELECT *  
FROM table-name1  
NATURAL JOIN table-name2;
```

- **Example**

```
SELECT *  
FROM class  
NATURAL JOIN class_info;
```

class

ID	Name
1	Abhi
2	Alish
4	Aayush

class_info

ID	Address
1	DHG
2	BHR
3	KTM

Result

ID	Name	ID	Address
1	Abhi	1	DHG
2	Alish	2	BHR

OUTER JOIN

- **OUTER JOIN** is based on both **matched** and **unmatched** data.
- Types of Outer JOIN
 - **LEFT OUTER JOIN**
 - **RIGHT OUTER JOIN**
 - **FULL OUTER JOIN**

LEFT OUTER JOIN

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

- **Syntax**

```
SELECT column-name-list  
FROM table-name1  
LEFT OUTER JOIN table-name2  
ON table-name1.column-name = table-name2.column-name;
```

Example

```
SELECT *  
FROM class  
LEFT OUTER JOIN class_info  
ON class.ID = class_info.ID;
```

class

ID	Name
1	Abhi
2	Alish
3	Aayush
4	Anu
5	Ashish

class_info

ID	Address
1	DHG
2	BHR
3	KTM
7	NPJ
9	PKR

Result

ID	Name	ID	Address
1	Abhi	1	DHG
2	Alish	2	BHR
3	Aayush	3	BKT
4	Anu	Null	Null
5	Ashish	Null	Null

RIGHT OUTER JOIN

- The **RIGHT OUTER JOIN** returns
 - a result-set table with the **matched data from the two tables**,
 - the **remaining rows of the right table** and
 - **null from the left table's columns**.

- **Syntax**

```
SELECT column-name-list  
FROM table-name1  
RIGHT OUTER JOIN table-name2  
ON table-name1.column-name = table-name2.column-name;
```

Example

```
SELECT *  
FROM class  
RIGHT OUTER JOIN class_info  
ON class.ID = class_info.ID;
```

class

ID	Name
1	Abhi
2	Alish
3	Aayush
4	Anu
5	Ashish

class_info

ID	Address
1	DHG
2	BHR
3	KTM
7	NPJ
9	PKR

Result

ID	Name	ID	Address
1	Abhi	1	DHG
2	Alish	2	BHR
3	Aayush	3	BKT
Null	Null	7	NPJ
Null	Null	8	PKR

FULL OUTER JOIN

- The **FULL OUTER JOIN** returns
 - a result-set table with the **matched data from the two tables**,
 - the **remaining rows of the left table** and
 - the **remaining rows of the right table**.

- **Syntax**

```
SELECT column-name-list  
FROM table-name1  
FULL OUTER JOIN table-name2  
ON table-name1.column-name = table-name2.column-name;
```

class

ID	Name
1	Abhi
2	Alish
3	Aayush
4	Anu
5	Ashish

class_info

ID	Address
1	DHG
2	BHR
3	KTM
7	NPJ
9	PKR

Example

```
SELECT *  
FROM class  
FULL OUTER JOIN class_info  
ON class.ID = class_info.ID;
```

Result

ID	Name	ID	Address
1	Abhi	1	DHG
2	Alish	2	BHR
3	Aayush	3	BKT
4	Anu	Null	Null
5	Ashish	Null	Null
Null	Null	7	NPJ
Null	Null	8	PKR

Views

- Views are virtual or logical tables of data extracted from existing table. It allows users to structure data in a way that users or classes of users find natural or intuitive.
- It summarize data from various tables which can be used to generate report.
- View can be created from one or many tables which depends on the written SQL query to create a view.
- It doesnot require any disk space.

- **Syntax**

```
CREATE VIEW view_name  
AS  
<SELECT Query>
```

Example

```
CREATE VIEW sale_view  
AS  
SELECT * FROM sales  
WHERE customer = 'Anu';
```

o_id	order_name	previous_balance	customer
11	O1	2000	Abhi
12	O2	1000	Alish
13	O3	2000	Aayush
14	O4	1000	Anu
15	O5	2000	Ashish

Displaying View

```
SELECT * FROM sale_view;
```

o_id	order_name	previous_balance	customer
14	O4	1000	Anu

Update a VIEW

- UPDATE command for view is same as for tables.
- **Syntax**

UPDATE view-name SET VALUE [WHERE condition];

NOTE: If we update a view ,it also updates base table data automatically.

Read-Only VIEW

- We can create a view with read-only option to restrict access to the view.
- **Syntax**

CREATE or REPLACE VIEW view_name

AS

SELECT column_name(s) FROM table_name WHERE condition

WITH read-only;

Types of View

- There are two types of view :

Simple view

Complex view

Simple View

- created from single table
- does not contain functions and groups of data
- DML operation can be performed
- does not include NOT NULL columns from base table.
- **Syntax**

Create VIEW ViewName AS SELECT column1,...columnN From tableName;

Complex View

- created by using more than one tables.
- There should be relation between 2 tables to create complex view.
- It contains functions and group of data.
- DML operations could not always be performed through complex view.
- NOT NULL columns are included.

Example :

Consider tables:

Employee (emp_name, emp_num, dept_code)

Department(dept_code, dept_name)

To create complex view:

```
CREATE VIEW Emp_View
```

```
AS
```

```
SELECT e. emp_name, d.dept_name
```

```
FROM Employee as e
```

```
INNER JOIN Department as d
```

```
ON e.dept_code = d.dept_code;
```

Complex View

- created by using more than one tables.
- There should be relation between 2 tables to create complex view.
- It contains functions and group of data.
- DML operations could not always be performed through complex view.
- NOT NULL columns are included.

Example :

Consider tables:

Employee (emp_name, emp_num, dept_code)

Department(dept_code, dept_name)

To create complex view:

```
CREATE VIEW Emp_View
```

```
AS
```

```
SELECT e. emp_name, d.dept_name
```

```
FROM Employee as e
```

```
INNER JOIN Department as d
```

```
ON e.dept_code = d.dept_code;
```

Materialized View

- similar to a regular view in that it represents the result of a stored query
- unlike a regular view, a Materialized View stores the result of that query in a physical table
- a snapshot of the data at the time the MV is created or refreshed

Syntax

1. Create MV

```
CREATE MATERIALIZED VIEW view_name AS  
SELECT ...  
FROM ...  
WHERE ...;
```

2. Refresh MV

```
REFRESH MATERIALIZED VIEW view_name;
```

Common Table Expression(CTE)

- temporary named result sets that exist for just one query
- can be referenced within a SELECT, INSERT, UPDATE, or DELETE statement
- can be self-referencing or even recursive
- defined using the WITH clause

Why are CTEs used?

- Improved readability, better visibility
- Decomposing Complicated Queries
- Recursive Queries
- Referencing the Same Dataset Multiple Times

WITH Clause

- **WITH Clause allows you to give a sub-query block a name.**
- **This block can be referenced in several places.**
- **This is used for defining a temporary relation and can be used by the query that is associated with WITH Clause.**
- **Syntax:**

```
WITH cte_name AS (  
    -- CTE query here  
)  
-- Main query using the CTE  
SELECT ...  
FROM cte_name ...
```

Window Function

- a type of function that performs a calculation across a set of table rows that are somehow related to the current row
- this "set of related rows" is termed as a "window"
- Aka windowing functions, OVER functions or analytics functions

Why use window functions?

- Aggregation Without Grouping
- Flexible Calculations
- Complex Data Analysis Made Simpler

Window functions

- Aggregation functions: SUM(), AVG(), COUNT(), MIN(), MAX()
- Ranking functions: ROW_NUMBER(), RANK(), DENSE_RANK(), NTILE(n)
- Value functions: LAG(), LEAD(), FIRST_VALUE(), LAST_VALUE(), NTH_VALUE()
- Statistical functions
- Cumulative distribution and percentile functions

Basic windowing syntax

```
SELECT column_name1,  
    window_function(column_name2)  
    OVER([PARTITION BY column_name1] [ORDER BY column_name3]) AS new_column  
FROM table_name;
```

window_function= any aggregate or ranking function

column_name1= column to be selected

column_name2= column on which window function is to be applied

column_name3= column on whose basis partition of rows is to be done

new_column= Name of new column

table_name= Name of table

Over Clause

- defines the window function
- determines the range or set of rows used in the window function's calculation for each row
 - Without OVER: Aggregate functions like SUM, AVG, and COUNT return a single value for the entire set of rows.
 - With OVER: These same functions return a value for each row based on the window of rows defined by the OVER clause.

Partition By

- divides the query result set into partitions
- the window function is applied separately to each partition
- similar to the GROUP BY clause, but, instead of aggregating the data, it retains the original rows and computes the function over each partition
- PARTITION BY column_name
- only those columns made available by the FROM clause can be used in PARTITION BY
- aliases in the SELECT list can't be used for partition

Order By

- within the defined window, rows can be ordered using this clause
- i.e. it defines the logical order of rows within each partition of the result set
- crucial while calculating running totals or cumulative metrics
- if not specified, default order is ASC & the window function will use all rows in the partition
- like PARTITION BY, ORDER BY can also use only those columns specified by the FROM clause
- integer can't be specified to represent column name or alias

Frame Specification

Determines the range of rows to include in the calculation for the current row

Syntax components:

1. ROWS or RANGE

- ROWS defines the frame by a number of rows
- RANGE defines the frame by value intervals

2. Start and End boundaries

- UNBOUNDED PRECEDING: from the first row of the partition
- 'X' PRECEDING: 'x' number of rows before the current row
- CURRENT ROW: current row
- 'X' FOLLOWING: 'x' number of rows after the current row
- UNBOUNDED FOLLOWING: from the last row (till the current or preceding row)

Rows Preceding

```
SELECT employee_id, employee_name, quarter, year, sales_amount,  
       SUM(sales_amount) OVER (PARTITION by employee_id  
                                ORDER BY sales_amount  
                                ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) as CurrentAndPrev2  
FROM employeeperformance  
ORDER BY employee_id;
```

The screenshot displays a database query result in a table format. The table has columns for employee_id, employee_name, quarter, year, sales_amount, and a calculated column named currentandprev2. The data is ordered by employee_id. Blue arrows indicate the rows preceding the current row in the window function calculation.

	employee_id integer	employee_name character varying (50)	quarter character varying (10)	year integer	sales_amount numeric	currentandprev2 numeric
1	101	Alice	Q2	2023	20000	20000
2	101	Alice	Q3	2023	20000	40000
3	101	Alice	Q4	2023	22000	62000
4	101	Alice	Q1	2024	23000	65000
5	101	Alice	Q2	2024	25000	70000
6	102	Bob	Q4	2023	15000	15000
7	102	Bob	Q1	2024	15000	30000
8	102	Bob	Q2	2024	30000	60000
9	103	Charlie	Q3	2023	25000	25000
10	103	Charlie	Q4	2023	28000	53000
11	103	Charlie	Q1	2024	28000	81000
12	103	Charlie	Q2	2024	30000	86000
13	104	Danny	Q1	2024	22000	22000
14	104	Danny	Q2	2024	22000	44000

Rows Following

```
SELECT employee_id, employee_name, quarter, year, sales_amount,  
       SUM(sales_amount) OVER (PARTITION by employee_id  
                                ORDER BY sales_amount  
                                ROWS BETWEEN CURRENT ROW AND 2 FOLLOWING) as CurrentandFollow2  
FROM employeeperformance  
ORDER BY employee_id;
```

	employee_id integer	employee_name character varying (50)	quarter character varying (10)	year integer	sales_amount numeric	currentandfollow2 numeric
1	101	Alice	Q2	2023	20000	62000
2	101	Alice	Q3	2023	20000	65000
3	101	Alice	Q4	2023	22000	70000
4	101	Alice	Q1	2024	23000	48000
5	101	Alice	Q2	2024	25000	25000
6	102	Bob	Q4	2023	15000	60000
7	102	Bob	Q1	2024	15000	45000
8	102	Bob	Q2	2024	30000	30000
9	103	Charlie	Q3	2023	25000	81000
10	103	Charlie	Q4	2023	28000	86000
11	103	Charlie	Q1	2024	28000	58000
12	103	Charlie	Q2	2024	30000	30000
13	104	Danny	Q1	2024	22000	44000
14	104	Danny	Q2	2024	22000	22000

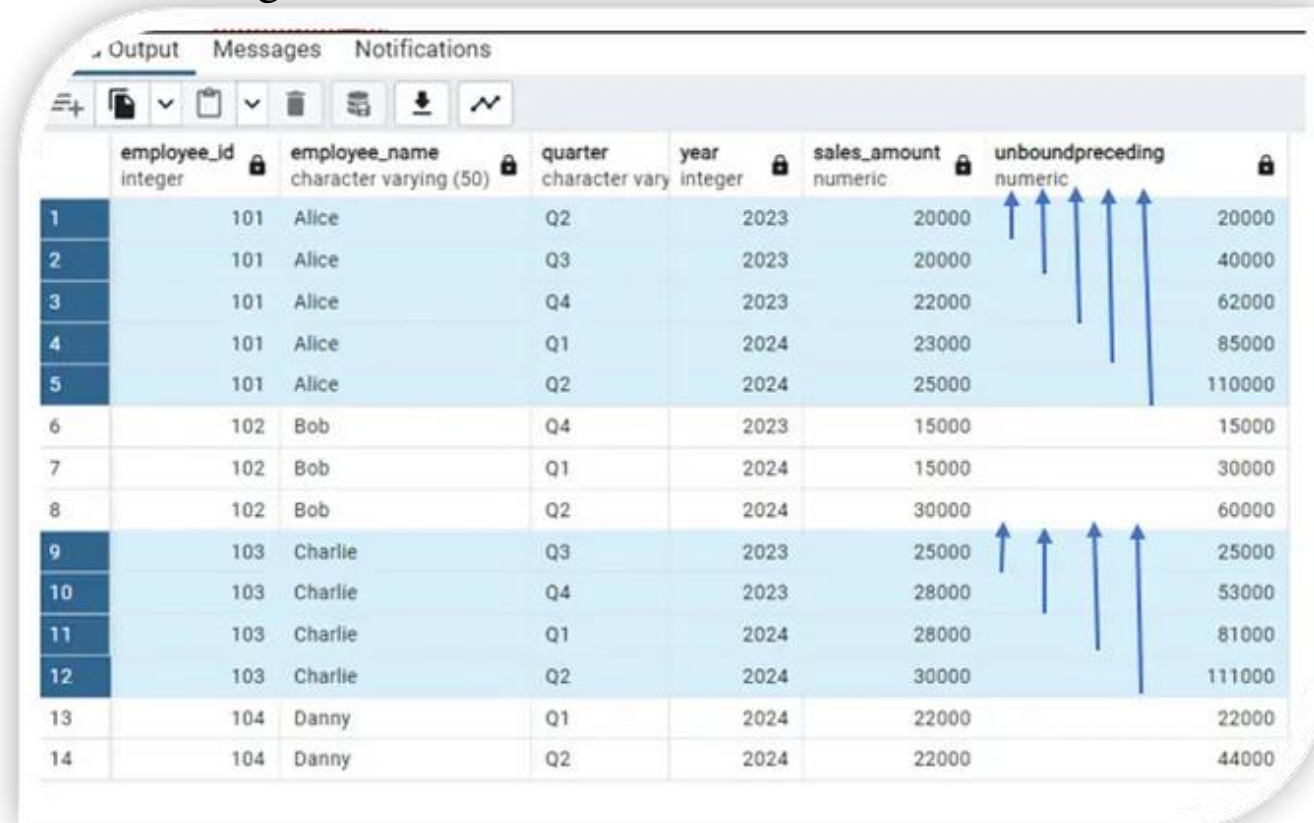
Both Preceding and Following

```
SELECT employee_id, employee_name, quarter, year, sales_amount,  
       SUM(sales_amount) OVER (PARTITION by employee_id  
                                ORDER BY sales_amount  
                                ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) as BeforeandAfter  
FROM employeeperformance  
ORDER BY employee_id;
```

	employee_id integer	employee_name character varying (50)	quarter character varying (10)	year integer	sales_amount numeric	beforeandafter numeric
1	101	Alice	Q2	2023	20000	40000
2	101	Alice	Q3	2023	20000	62000
3	101	Alice	Q4	2023	22000	65000
4	101	Alice	Q1	2024	23000	70000
5	101	Alice	Q2	2024	25000	48000
6	102	Bob	Q4	2023	15000	30000
7	102	Bob	Q1	2024	15000	60000
8	102	Bob	Q2	2024	30000	45000
9	103	Charlie	Q3	2023	25000	53000
10	103	Charlie	Q4	2023	28000	81000
11	103	Charlie	Q1	2024	28000	86000
12	103	Charlie	Q2	2024	30000	58000
13	104	Danny	Q1	2024	22000	44000
14	104	Danny	Q2	2024	22000	44000

UNBOUNDED PRECEDING

```
SELECT employee_id, employee_name, quarter, year, sales_amount,  
       SUM(sales_amount) OVER (PARTITION by employee_id  
                                ORDER BY sales_amount  
                                ROWS UNBOUNDED PRECEDING) as UnboundPreceding  
FROM employeeperformance  
ORDER BY employee_id;
```



The screenshot displays a database query result with columns: employee_id, employee_name, quarter, year, sales_amount, and unboundpreceding. The data is ordered by employee_id. The 'unboundpreceding' column shows the cumulative sum of sales_amount for each employee partition, with arrows indicating the calculation from the first row of each partition to the current row.

	employee_id	employee_name	quarter	year	sales_amount	unboundpreceding
1	101	Alice	Q2	2023	20000	20000
2	101	Alice	Q3	2023	20000	40000
3	101	Alice	Q4	2023	22000	62000
4	101	Alice	Q1	2024	23000	85000
5	101	Alice	Q2	2024	25000	110000
6	102	Bob	Q4	2023	15000	15000
7	102	Bob	Q1	2024	15000	30000
8	102	Bob	Q2	2024	30000	60000
9	103	Charlie	Q3	2023	25000	25000
10	103	Charlie	Q4	2023	28000	53000
11	103	Charlie	Q1	2024	28000	81000
12	103	Charlie	Q2	2024	30000	111000
13	104	Danny	Q1	2024	22000	22000
14	104	Danny	Q2	2024	22000	44000

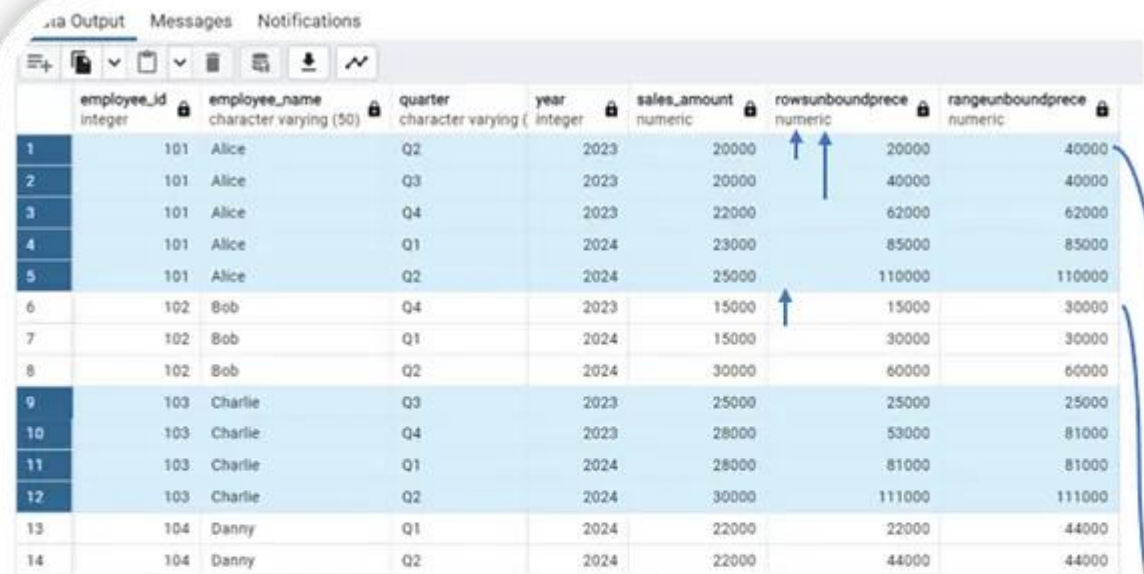
UNBOUNDED FOLLOWING

```
SELECT employee_id, employee_name, quarter, year, sales_amount,  
       SUM(sales_amount) OVER (PARTITION by employee_id  
                               ORDER BY sales_amount  
                               ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING ) as UnboundFollowing  
FROM employeeperformance  
ORDER BY employee_id;
```

	employee_id integer	employee_name character varying (50)	quarter character vary	year integer	sales_amount numeric	unboundfollowing numeric
1	101	Alice	Q2	2023	20000	110000
2	101	Alice	Q3	2023	20000	90000
3	101	Alice	Q4	2023	22000	70000
4	101	Alice	Q1	2024	23000	48000
5	101	Alice	Q2	2024	25000	25000
6	102	Bob	Q4	2023	15000	60000
7	102	Bob	Q1	2024	15000	45000
8	102	Bob	Q2	2024	30000	30000
9	103	Charlie	Q3	2023	25000	111000
10	103	Charlie	Q4	2023	28000	86000
11	103	Charlie	Q1	2024	28000	58000
12	103	Charlie	Q2	2024	30000	30000
13	104	Danny	Q1	2024	22000	44000
14	104	Danny	Q2	2024	22000	22000

RANGE Framing

```
SELECT employee_id, employee_name, quarter, year, sales_amount,  
       SUM(sales_amount) OVER (PARTITION by employee_id  
       ORDER BY sales_amount  
       ROWS UNBOUNDED PRECEDING ) as RowsUnboundprece,  
       SUM(sales_amount) OVER (PARTITION by employee_id  
       ORDER BY sales_amount  
       RANGE UNBOUNDED PRECEDING ) as RangeUnboundprece  
FROM employeeperformance  
ORDER BY employee_id;
```



	employee_id integer	employee_name character varying (50)	quarter character varying (5)	year integer	sales_amount numeric	rowsunboundprece numeric	rangeunboundprece numeric
1	101	Alice	Q2	2023	20000	20000	40000
2	101	Alice	Q3	2023	20000	40000	40000
3	101	Alice	Q4	2023	22000	62000	62000
4	101	Alice	Q1	2024	23000	85000	85000
5	101	Alice	Q2	2024	25000	110000	110000
6	102	Bob	Q4	2023	15000	15000	30000
7	102	Bob	Q1	2024	15000	30000	30000
8	102	Bob	Q2	2024	30000	60000	60000
9	103	Charlie	Q3	2023	25000	25000	25000
10	103	Charlie	Q4	2023	28000	53000	81000
11	103	Charlie	Q1	2024	28000	81000	81000
12	103	Charlie	Q2	2024	30000	111000	111000
13	104	Danny	Q1	2024	22000	22000	44000
14	104	Danny	Q2	2024	22000	44000	44000

It starts at the first row of the partition and extends up to the current row. However, it includes all rows that have the same value as the current row for the columns specified in the ORDER BY clause, in addition to all preceding rows. (Here ORDER BY sales_amount)