

SQL series



# Advanced SQL Concepts

**With Code Examples**

Advanced Level

# Advanced SQL Concepts

This presentation will cover several advanced SQL topics, including **Window Functions, Recursive Queries with CTEs, Advanced Joins and Subqueries, and more.**

These concepts are essential for working with complex data and solving intricate problems using SQL.

Swipe next →

# Window Functions

Window Functions are a powerful feature in SQL that allows you to perform calculations across a set of rows related to the current row. They provide a way to compute running totals, moving averages, rankings, and other analytical calculations.

```
CREATE TABLE sales_data (  
    product_name VARCHAR(100),  
    category VARCHAR(50),  
    sales DECIMAL(10, 2)  
);  
  
INSERT INTO sales_data (product_name, category, sales)  
VALUES  
    ('Product A', 'Category 1', 1000.00),  
    ('Product B', 'Category 1', 2000.00),  
    ('Product C', 'Category 2', 1500.00),  
    ('Product D', 'Category 2', 2500.00);  
  
FROM  
    sales_data;
```

# Example

This query calculates the total sales for each category using the SUM window function with the PARTITION BY clause.

```
SELECT
    product_name,
    category,
    sales,
    SUM(sales) OVER (PARTITION BY category) AS category_total_sales
FROM
    sales_data;
```

follow for more

# Recursive Queries with CTEs

```
CREATE TABLE employees (  
    employee_id INT PRIMARY KEY,  
    employee_name VARCHAR(100),  
    manager_id INT,  
    FOREIGN KEY (manager_id) REFERENCES employees(employee_id)  
);  
  
INSERT INTO employees (employee_id, employee_name, manager_id)  
VALUES  
    (1, 'John Doe', NULL),  
    (2, 'Jane Smith', 1),  
    (3, 'Bob Johnson', 1),  
    (4, 'Alice Williams', 2),  
    (5, 'Tom Brown', 3);
```

Swipe next →

# Example

This recursive CTE traverses the employee hierarchy, starting from the top-level managers, and retrieves all employees with their respective levels in the hierarchy.

```
WITH RECURSIVE employee_hierarchy AS (  
    SELECT  
        employee_id,  
        manager_id,  
        employee_name,  
        1 AS level  
    FROM  
        employees  
    WHERE  
        manager_id IS NULL  
    UNION ALL  
    SELECT  
        e.employee_id,  
        e.manager_id,  
        e.employee_name,  
        eh.level + 1  
    FROM  
        employees e  
        INNER JOIN employee_hierarchy eh ON e.manager_id = eh.employee_id  
)  
SELECT  
    *  
FROM  
    employee_hierarchy;
```

# Advanced Joins and Subqueries

```
CREATE TABLE products (  
    product_id INT PRIMARY KEY,  
    product_name VARCHAR(100),  
    category_id INT,  
    FOREIGN KEY (category_id) REFERENCES categories(category_id)  
);  
  
CREATE TABLE categories (  
    category_id INT PRIMARY KEY,  
    category_name VARCHAR(50)  
);  
  
CREATE TABLE sales_data (  
    sale_id INT PRIMARY KEY,  
    product_id INT,  
    sales DECIMAL(10, 2),  
    FOREIGN KEY (product_id) REFERENCES products(product_id)  
);  
  
INSERT INTO categories (category_id, category_name)  
VALUES  
    (1, 'Category 1'),  
    (2, 'Category 2');  
  
INSERT INTO products (product_id, product_name, category_id)  
VALUES  
    (1, 'Product A', 1),  
    (2, 'Product B', 1),  
    (3, 'Product C', 2),  
    (4, 'Product D', 2);  
  
INSERT INTO sales_data (sale_id, product_id, sales)  
VALUES  
    (1, 1, 1000.00),  
    (2, 1, 500.00),  
    (3, 2, 2000.00),  
    (4, 3, 1500.00),  
    (5, 4, 2500.00);
```

Swipe next →

# Example

This query combines data from the products and categories tables using an inner join and calculates the total sales for each product using a correlated subquery.

```
SELECT
  p.product_name,
  c.category_name,
  (
    SELECT
      SUM(sales)
    FROM
      sales_data sd
    WHERE
      sd.product_id = p.product_id
  ) AS total_sales
FROM
  products p
  INNER JOIN categories c ON p.category_id = c.category_id;
```



follow for more

# Window Functions: Ranking Functions

This query ranks the products within each category based on their sales, using the RANK window function. (Use the sales\_data table)

```
SELECT
    product_name,
    category,
    sales,
    RANK() OVER (PARTITION BY category ORDER BY sales DESC) AS rank_by_sales
FROM
    sales_data;
```

Swipe next →

# Recursive Queries: Generating Hierarchical Data

This recursive CTE generates a calendar table with dates from January 1, 2023, to December 31, 2023.

```
WITH RECURSIVE calendar AS (  
    SELECT  
        CAST('2023-01-01' AS DATE) AS date_value,  
        1 AS level  
    UNION ALL  
    SELECT  
        DATEADD(DAY, 1, date_value),  
        level + 1  
    FROM  
        calendar  
    WHERE  
        date_value < '2023-12-31'  
)  
SELECT  
    date_value  
FROM  
    calendar  
ORDER BY  
    date_value;
```

# Advanced Joins: Self-Joins

This query uses a self-join on the employees table to retrieve the employee names and their corresponding manager names. (Use the employees table from Slide 3.)

```
SELECT
  e1.employee_name AS employee,
  e2.employee_name AS manager
FROM
  employees e1
  INNER JOIN employees e2 ON e1.manager_id = e2.employee_id;
```

# Subqueries in the FROM Clause

This query uses a subquery in the FROM clause to retrieve the product name, category, and sales data, and then calculates the total sales for each category. (Use the sales\_data table)

```
SELECT
    category,
    SUM(sales) AS total_sales
FROM
    (
        SELECT
            product_name,
            category,
            sales
        FROM
            sales_data
        ) AS product_sales
GROUP BY
    category;
```

follow for more

# Advanced Analytic Function

This query demonstrates the use of the **LEAD**, **LAG**, **FIRST\_VALUE**, **LAST\_VALUE**, and **NTH\_VALUE**, which allow you to perform complex data analysis and calculations based on the ordering and partitioning of rows.

```
CREATE TABLE stock_prices (  
    stock_symbol VARCHAR(10),  
    trade_date DATE,  
    open_price DECIMAL(10, 2),  
    close_price DECIMAL(10, 2)  
);  
  
INSERT INTO stock_prices (stock_symbol, trade_date, open_price, close_price)  
VALUES  
    ('ACME', '2023-04-01', 100.00, 102.50),  
    ('ACME', '2023-04-02', 103.00, 101.75),  
    ('ACME', '2023-04-03', 102.25, 104.00),  
    ('BXYZ', '2023-04-01', 50.00, 51.25),  
    ('BXYZ', '2023-04-02', 51.50, 52.00),  
    ('BXYZ', '2023-04-03', 51.75, 50.50);
```

Swipe next →

# Example

```
-- LAG example
SELECT
    stock_symbol,
    trade_date,
    open_price,
    close_price,
    LAG(open_price, 1) OVER (PARTITION BY stock_symbol ORDER BY trade_date)
AS previous_day_open
FROM
    stock_prices;

-- LEAD example (same as in the previous slide)
SELECT
    stock_symbol,
    trade_date,
    open_price,
    close_price,
    LEAD(close_price, 1) OVER (PARTITION BY stock_symbol ORDER BY
trade_date) AS next_day_close
FROM
    stock_prices;


-- FIRST_VALUE and LAST_VALUE examples
SELECT
    stock_symbol,
    trade_date,
    open_price,
    close_price,
    FIRST_VALUE(open_price) OVER (PARTITION BY stock_symbol ORDER BY
trade_date) AS first_open_price,
    LAST_VALUE(close_price) OVER (PARTITION BY stock_symbol ORDER BY
trade_date ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS
last_close_price
FROM
    stock_prices;

-- NTH_VALUE example
SELECT
    stock_symbol,
    trade_date,
    open_price,
    close_price,
    NTH_VALUE(close_price, 2) OVER (PARTITION BY stock_symbol ORDER BY
trade_date ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS
second_close_price
FROM
    stock_prices;
```

Swipe next →

# Recursive Queries - Generating Sequences

Recursive queries can be used to generate sequences of numbers or dates, which can be useful for various purposes, such as generating test data or handling gaps in sequences.



```
WITH RECURSIVE number_sequence AS (  
    SELECT 1 AS num  
    UNION ALL  
    SELECT num + 1  
    FROM number_sequence  
    WHERE num < 100  
)  
SELECT num  
FROM number_sequence;
```

This recursive CTE generates a sequence of numbers from 1 to 100.



# Correlated Subqueries


Correlated subqueries are subqueries that reference columns from the outer query. They can be used to perform complex filtering or calculations based on data from the outer query.

```
CREATE TABLE orders (  
  order_id INT PRIMARY KEY,  
  customer_id INT,  
  order_date DATE,  
  total_amount DECIMAL(10, 2)  
);  
  
INSERT INTO orders (order_id, customer_id, order_date, total_amount)  
VALUES  
  (1, 1, '2023-03-01', 100.00),  
  (2, 1, '2023-03-15', 200.00),  
  (3, 2, '2023-03-20', 150.00),  
  (4, 2, '2023-04-01', 300.00);
```



# Example

While OOP offers many benefits, it's essential to be mindful of potential pitfalls, such as complexity due to deep inheritance hierarchies, tight coupling between classes, and misuse of design patterns. Striking the right balance and following best practices is crucial for maintainable and scalable code.



```
SELECT
    customer_id,
    order_id,
    order_date,
    total_amount,
    (
        SELECT MAX(total_amount)
        FROM orders o2
        WHERE o2.customer_id = o1.customer_id
              AND o2.order_date < o1.order_date
    ) AS previous_max_order
FROM
    orders o1;
```

# Pitfalls and Best Practices

While advanced SQL concepts provide powerful capabilities, it's important to be aware of potential pitfalls and follow best practices:

- Optimize queries for performance, especially when dealing with large datasets or complex operations.
- Ensure data integrity and consistency when using recursive queries or hierarchical structures.
- Test thoroughly and validate results, especially when working with complex queries.
- Consider using database views or stored procedures for code organization and maintainability.
- Document your SQL code for better collaboration and future reference.

# Advanced SQL Concepts to Improve Your and Code's Performance

*In this article, we'll discuss advanced SQL concepts and their use cases. Then we'll use them to answer interview questions from reputable employers.*



SQL is an essential skill for working with databases. It allows beginner data scientists to easily select, filter, and aggregate data. You can even combine some of these basic SQL features to perform more difficult tasks.

SQL also has more advanced features, which allow you to perform difficult tasks using fewer lines of code. In this article, we'll discuss useful but lesser-known advanced SQL features for writing more efficient queries.

We will also discuss advanced use cases for familiar SQL features. For example, how to use GROUP BY statements to calculate subtotals.

# Concepts to Write Advanced SQL Queries



Let's take a look at some of the most useful advanced SQL concepts and their use cases.

## JOINS

JOINS are among the essential SQL features, often used to combine tables based on relationships between them. Normally, JOINS do not increase the volume of records, only the breadth of information.

In practice, you will often encounter cases when necessary information is stored in two different tables. You can use JOINS to 'fill in' the gaps and gain access to data in both tables.

Regardless of your role or seniority level, you need to know the basic syntax for writing JOINS and the ON clause, which is used to specify the shared dimension and additional conditions. Depending on your specialty, you may need to learn beyond the basics.

For example, you may need to chain multiple JOINS to combine three, four, or even more tables. Sometimes you'll need to chain [different types of JOINS](#).

Strong knowledge of JOINS will also help you remove unnecessary complexity. For example, using the ON clause to do all the filtering (when possible).

## DISTINCT

DISTINCT clause is the easiest way to remove duplicate values from a column. It's not exactly an advanced SQL concept, but DISTINCT can be combined with other features (like aggregate functions). For example, it's the most efficient way to **COUNT()** only unique values in a certain column.

Many data scientists don't know that the DISTINCT clause considers every NULL value to be unique. When applied to multiple columns, DISTINCT only returns unique pairs of values.

## Set Operators

Set operators are simple but essential tools for working with datasets. They allow you to vertically combine multiple SELECT statements into one result, find common rows between datasets, or rows that appear in one dataset but not the other.

To use set operators, you should follow simple rules - datasets (the result of a query statement) must have the same number and type of columns.

Let's start with the most useful and common set operators - UNION and UNION ALL. These allow you to combine data vertically. In other words, stack the results of queries on top of one another. The big difference is that UNION removes duplicate rows, UNION ALL doesn't.

UNIONs increase the overall number of rows in the table.

You can use UNIONs to vertically combine the result of complex queries, not just the result of a simple SELECT statement. As long as datasets satisfy conditions - they have the same number and type of columns.

PostgreSQL also supports the INTERSECT operator, which returns the intersection of specified datasets. In other words, common rows that appear in the results of queries. INTERSECT also removes duplicates. If the same row appears across datasets multiple times, INTERSECT will return only one row.

MySQL also supports MINUS. It returns rows from the first dataset that do not appear in subsequent datasets. It is not supported in PostgreSQL, but you can use the EXCEPT operator to the same effect.

## Subqueries and Common Table Expressions (CTEs)

Subqueries are a common SQL feature, often used to work with the result of a certain query. Some subqueries have subqueries of their own - these are called nested subqueries.

SQL experts often use subqueries to set a condition. More specifically, use them as a list of acceptable values for the IN or SOME logical operators. Later on, we'll explore this use case in practice.

If you have nested subqueries or subqueries with complex logic, it's better to define that subquery separately, and store its result as a common table expression. Then you can reference its result in the main query instead of embedding the entire subquery.

## Aggregate Functions

Aggregate functions are essential for doing calculations in SQL. For example - counting the number of rows, finding the highest and lowest values, calculating totals, and so on. There are dozens of aggregate functions in SQL, but the five most important are: **SUM()**, **COUNT()**, **AVG()**, **MIN()**, and **MAX()**.

Aggregate functions have many different use cases. When applied to groups, they aggregate values in the group. Without a GROUP BY statement, they aggregate values in the entire table. Also, **COUNT(\*)** returns the number of all rows, but **COUNT(column)** only counts non-NULL values.

Aggregate functions are very versatile and can be combined with other SQL features. For example, in the following sections, we'll see an example of using the **AVG()** function with a CASE statement. Aggregate functions are also often paired with the DISTINCT clause.

Our blog post on [SQL Aggregate Functions](#) takes a closer look at this feature and gives you plenty of opportunities to practice aggregate functions.

For a full list of aggregate functions, read [PostgreSQL documentation](#).

Later on, we'll use advanced SQL concepts to answer actual interview questions, including lesser-known aggregate functions like **CORR()**, which calculates correlation.

You should know how to use the AS command to name the result of aggregate functions.

# Window Functions

Window functions, also called analytical functions, are among the most useful SQL features. They are very versatile and often the most efficient tool for performing difficult and specific tasks in SQL.

Learning to use window functions comes down to understanding what each function does, general window function syntax, using the OVER clause, partitions, and window frames to specify the input, and ordering records within the input.

There are many different window functions, separated into three categories:

1. Aggregate window functions
2. Navigation functions
3. Ranking window functions

We'll go over each category of window functions in detail.

In window functions, we use the OVER clause to define a set of related records. For example, if we want to find an average salary for each department, we'd use OVER and PARTITION BY clauses to apply the **AVG()** function to records with the same **department** value.

PARTITION BY subclause, used with the OVER clause, creates partitions. In principle, partitions are similar to groups. From the previous example, records would be partitioned by **department** value. The number of rows won't change. The window function will create a new column to store the result.

Most data scientists know how to use PARTITION BY to narrow down inputs to the window functions. What they don't know is that you can use window frames to specify the input further.

Sometimes you need to apply window functions to only a subset of a partition. For example, you need to create a new value - the average salary of the TOP 10 earners in each department. You'll need window frames to only apply window functions to the first 10 records in the partition (assuming they are ordered in descending order by salary).

You can learn more about window function frames by reading our blog post on [SQL Window Functions](#).

## Aggregate Window Functions

The result of normal aggregate functions is collapsed into a single row. When applied to groups, one row contains aggregate results for each group.

You can use the same functions as window functions. This time, the result is not collapsed into one row. All rows stay intact, and the result of the aggregate function is stored in a separate column.

To use aggregate window functions, you need the **OVER** clause to define the window frame. **PARTITION BY** is optional but commonly used to specify the input further.

## Navigation Window Functions

Navigation functions (often also called value functions) are also extremely useful. They generate a value based on the value of another record. Let's look at a specific example.

In your day-to-day job as a data scientist, an employer might ask you to calculate year-over-year (also day-to-day or any other time frame) changes in metrics like growth, decline, churn, and so on. Let's say you are given sales in 12 months of the year and tasked with calculating monthly changes in sales.

There are multiple ways to calculate monthly sales growth (or decline). The easiest and most efficient way is to use the **LAG()** function, which returns a value (total sales) from the previous row. This way, you'll have sales for the current and previous months in one row. Then you can easily find the difference and calculate the percentage.

**LAG()** is one of the advanced SQL concepts that improve code performance.

The **LEAD()** function does the opposite - it allows you to access values from the following rows.

## Analytic Rank Functions

These functions allow you to create number values for each record. They're not the only way to generate number values, but rank functions are short and efficient.

Functions like **RANK()**, **DENSE\_RANK()**, and **PERCENT\_RANK()** rank records by value in the specified column. On the other hand, **ROW\_NUMBER()** assigns numbers to rows based on their position within the input.

You can also combine **ROW\_NUMBER()** with **ORDER BY** to rank records. We will explore this use case in practice later.

For a more detailed look, read our blog post on [SQL Rank Functions](#).



## Date-Time Manipulation

Date and time values are very common. Anyone working with data should be familiar with essential functions for date-time manipulation.

The **CURRENT\_DATE** function returns today's date. We will later use it to answer one of the interview questions.

We can use the **DATEADD()** function to add date values. The first argument to the function is the unit of date to add, the second is the number of units, and the third is the date to which we're adding.

**DATEDIFF()** function finds the difference between two dates and allows you to specify the unit to return. For example, if you specify 'month', it will return the difference between two dates in months. In that case, the first argument would be the unit of time, the second would be the start date, and the third would be the end date.

**DATENAME()** function returns a specified part of the date as a string. You can specify any date component - from year to weekday, and it will return that part of the date value. For the year, that may be '2017', for weekdays, it might be 'Sunday' or another day of the week.

The first argument is the part you want to return (as a string), and the second is the date value.

**YEAR()**, **MONTH()**, and **DAY()** functions extract year, month, and day components from the date value.

There is also a more general **EXTRACT()** function, which allows you to specify the part of the date you want to extract from date or timestamp values. The first argument to the function specifies the component that should be extracted - year, month, week, day, or even seconds, and the second specifies the date value from which you want to extract.

Check official [PostgreSQL documentation](#) to learn more about possible arguments to **EXTRACT()** and other functions on this list.

In some cases, you might need to cast values to (and from) date type. You can do this using the shorthand double semicolon (::) syntax or the **CAST()** function.

## CASE WHEN

In SQL, the CASE expression allows you to create an expression to define conditions and return specific values if that condition is met.

You should already be familiar with its basic use cases, such as labeling data based on the a condition. But you can also use CASE WHEN with aggregate functions, WHERE, HAVING, ORDER BY, and even GROUP BY statements.

For more information, read our blog post about [CASE WHEN statements in SQL](#).

## **GROUP BY and HAVING**

GROUP BY is a simple statement, but it has advanced features like CUBE and ROLLUP. These allow you to return totals as well as subtotals for each group.

For example, if you create groups for date and city values, ROLLUP will also return aggregate results for each date and each city.

The HAVING clause allows you to filter the result of aggregate results. It works like the WHERE clause - you need to specify the condition.

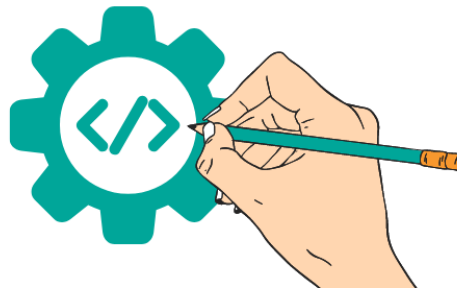
## **WHERE and Conditions**

This is a simple SQL feature to filter rows in a dataset. However, the conditions for the WHERE statement can be quite complex.

SQL allows you to use logical operators like IN, BETWEEN, NOT, EXISTS, and LIKE. You can use these operators to set complex conditions. For example, if you want to filter records by date value, you might use the BETWEEN logical operator to specify the acceptable range of dates.

The IN logical operator can be set to an expression that returns a list of values. It filters values in a column depending on whether or not it appears in the list.

# Advanced SQL Queries in Practice



Let's explore how to use the aforementioned SQL features to answer actual [SQL Interview Questions](#).

## Question 1: More Than 100 Dollars

"The company for which you work is reviewing their 2021 monthly sales.

For each month of 2021, calculate what percentage of restaurants that have reached atleast 100\$ or more in monthly sales.

Round your percentage to the nearest two decimal places.

Note: Please keep in mind that if an order has a blank value for `actual_delivery_time`, it has been cancelled and therefore does not count towards monthly sales."

Link to the question: <https://platform.stratascratch.com/coding/2115-more-than-100-dollars>

### Available data:

To solve this question, you need to work with data in two tables: **delivery\_orders** and **order\_values**.

Before answering the question, try to identify important values in each table. Let's start with **delivery\_orders**:

delivery_id	order_placed_time	predicted_delivery_time	actual_delivery_time	delivery_rating	driver_id	restaurant_id
O2132	2021-11-17 04:45:00	2021-11-17 05:37:00	2021-11-17 05:58:00	4	D239	R0001
O2152	2021-12-09 19:09:00	2021-12-09 19:41:00	2021-12-09 19:41:00	3	D238	R0001
O2158	2022-01-04 02:31:00	2022-01-04 02:56:00	2022-01-04 03:21:00	4	D239	R0001
O2173	2022-02-09 01:10:00	2022-02-09 01:10:00	2022-02-09 01:10:00	0	D239	R0001

```

delivery_id:      varchar
order_placed_time:  datetime
predicted_delivery_time:  datetime
actual_delivery_time:  datetime
delivery_rating:    float
driver_id:         varchar
restaurant_id:     varchar
consumer_id:       varchar

```

- **delivery\_id** values are the shared dimension. We'll use it to JOIN two tables.
- We need to find each restaurant's monthly sales. It's safe to assume that **order\_placed\_time** is going to be important.
- Most likely we won't need values like **predicted\_delivery\_time**, **actual\_delivery\_time**, **delivery\_rating**, **delivery\_rating**, or **dasher\_id**.
- We'll use the **restaurant\_id** value to aggregate sales for each restaurant.
- We don't need to identify customers who placed the order, so **consumer\_id** values can be ignored.

Now let's look at the **order\_value** table:

delivery_id	sales_amount
O2141	15.28
O2159	16.67
O2151	24.64
O2130	25.43
O2165	29.74

delivery_id:	varchar
sales_amount:	float

- We'll use **delivery\_id** values to JOIN two tables.
- **sales\_amount** values represent delivery value in dollars.

### Logical Approach

Solution to this question can be broken down into two steps:

1. Write a query that returns every restaurant's monthly revenue.
2. For every month, find the percentage of restaurants that had at least \$100 in sales.

To find every restaurant's monthly revenue, we need to work with information from two tables. One contains information about deliveries (who prepared food, who delivered, at what time, and so on), and another contains information like each order's dollar value. We'll definitely need to JOIN two tables.

Also, it's important to note that every row contains information about individual delivery. To get monthly sales for each restaurant, we'll need to aggregate records by month and **restaurant\_id**.

Unfortunately, the table does not have a month value. In the SELECT statement, we need to use the **EXTRACT()** function to get months from the **order\_placed\_time** value and save it. Then use the GROUP BY statement to create groups for every restaurant in every month.

The question also specifies that we only need to work with deliveries completed in 2021. So we need an additional WHERE statement to set the condition. Once again, we use the EXTRACT() function to get the year from the **order\_placed\_time** value and make sure that it is 2021.

We also need to use the **SUM()** aggregate function to find total sales for each restaurant in each month. We can simply save total sales for each restaurant, but it's much better if we use comparison operators (less than, or greater than) to compare the result of **SUM()** with 100 and save the result of a condition as a boolean.

In conclusion, a subquery that returns every restaurant's monthly revenue will involve a fair bit of logic. It's better to write this query outside of the main query, and save its result as a CTE.

With that, we have a query that calculates monthly sales for every restaurant.

To get the final answer, we'll calculate the percentage of restaurants that met this threshold in each month.

We need to create groups for every month and use the `AVG()` aggregate function to calculate the percentage of restaurants with higher than \$100 sales in that month.

As you may know, the `AVG()` function takes values from every record and then divides them by number of records to find the average. Our query does not contain total monthly sales for each restaurant, but a boolean that represents whether their sales were above 100.

Do you have an idea of how to use the **`AVG()`** function to find the ratio of **`true`** values in a certain column? Read our solution to find out how we did it.

## Write the code

### 1. JOIN two tables

Information necessary to solve this question is split between two tables, so we need to combine them.

**`delivery_id`** is the shared dimension between them.

```
SELECT *  
FROM delivery_orders d  
JOIN order_value v ON d.delivery_id = v.delivery_id
```

One table contains information about deliveries, the other - about delivery values.

Both tables contain essential information - we don't need delivery information without order values and vice versa. We should perform an `INNER JOIN`, which removes any deliveries without a match.

### 2. Filter deliveries by year

Tables contain delivery information for many years. The question explicitly tells us that we're only interested in restaurant sales in 2021.

```
SELECT *  
FROM delivery_orders d
```

```
JOIN order_value v ON d.delivery_id = v.delivery_id
WHERE EXTRACT(YEAR
              FROM order_placed_time) = 2021
```

Once we use INNER JOIN to combine two tables, we need to set an additional WHERE clause to filter deliveries by year.

There is no available data for the year when the order was placed, so we use the **EXTRACT()** function to get the year value from the **order\_placed\_time** value.

### 3. Find monthly sales for each restaurant

We use JOIN and WHERE to get information about all deliveries in 2021. Next, we must calculate monthly sales for each restaurant.

First, we use **EXTRACT()** to get the month from the **order\_placed\_time** value. This is the month when the order was placed. We use the **AS** command to give it a descriptive name.

Next, we use **GROUP BY** to group records by unique pairs of **restaurant\_id** and **month** values. Each group will have delivery records for each restaurant in each month.

Finally, we can use **SUM()** to add up **sales\_amount** values for each group.

Technically, we could save total sales for each restaurant and later determine what percentage of these values is higher than 100. However, it's better to get it out of the way now and simply store a boolean that represents whether monthly sales are above the \$100 minimum.

```
SELECT restaurant_id,
       EXTRACT(MONTH
              FROM order_placed_time) AS month,
       SUM(sales_amount) > 100 AS above_100
FROM delivery_orders d
JOIN order_value v ON d.delivery_id = v.delivery_id
WHERE EXTRACT(YEAR
              FROM order_placed_time) = 2021
GROUP BY restaurant_id,
         month
```

As you can see, the query will return three values - **restaurant\_id**, **month**, and **above\_100** boolean.

With that, we have a query that returns monthly sales for every restaurant. It's fairly complex, so we should save it as CTE and simply reference the result.

#### 4. Find the percentage of restaurants with over \$100 in sales in each month

Query from step 3 returns all the information we need: restaurant id, monthly sales for that restaurant, and an **above\_100** value which represents whether or not those restaurants made at least \$100 in sales. We will write it as a CTE.

Now we need to find what percentage of restaurants made more than \$100 in sales. In other words, in each month, what percentage of restaurant records have **above\_100** values that are **true**.

The question tells us to output the month and share of restaurants with above \$100 in sales. At this point, each record represents a single restaurant in a month.

We'll need to GROUP records by month and somehow calculate what percentage of **above\_100** values are **true**.

We can easily combine **AVG()** and CASE expressions to calculate this percentage.

The default behavior of **AVG()** is that it adds up all values in a group and divides the sum by the number of values to calculate the average. Instead, we can pass it a CASE expression that returns 1 if the **above\_100** is true and 0 if it's false.

If there are 10 restaurants in a month, and only 3 have more than 100\$ in sales, then **AVG()** will add up to 3, and divide it by 10, which gives us 0.3.

We multiply this ratio by 100 to get a percentage. As a cherry on top, we use the **AS** command to give it a descriptive name.

```
WITH cte AS
  (SELECT restaurant_id,
           EXTRACT(MONTH
                   FROM order_placed_time) AS month,
           SUM(sales_amount) > 100 AS above_100
   FROM delivery_orders d
   JOIN order_value v ON d.delivery_id = v.delivery_id
   WHERE EXTRACT(YEAR
                 FROM order_placed_time) = 2021
   GROUP BY restaurant_id,
            month)
SELECT month,
       100.0 * avg(CASE WHEN above_100 = True THEN 1 ELSE 0 END) AS
perc_over_100
FROM cte
```



GROUP BY month

This approach might seem unconventional, but CASE/WHEN is commonly used with aggregate functions.

Run the query to see if it matches the expected output for this question.

### Output

The query should return the percentage of restaurants with over 100\$ worth of sales in November and December.

month	pc
12	42.86
11	50

## Question 2: Completed Trip Within 168 Hours

“An event is logged in the events table with a timestamp each time a new rider attempts a signup (with an event name 'attempted\_su') or successfully signs up (with an event name of 'su\_success').

For each city and date, determine the percentage of signups in the first 7 days of 2022 that completed a trip within 168 hours of the signup date. HINT: driver id column corresponds to rider id column”

Link to the question:

<https://platform.stratascratch.com/coding/2134-completed-trip-within-168-hours>

### Available data:

We are given two tables. Let's take a quick look at values in the first:

rider_id	city_id	event_name	timestamp
r01	c001	su_success	2022-01-01 07:00:00
r02	c002	su_success	2022-01-01 08:00:00
r03	c002	su_success	2022-01-01 08:00:00
r04	c001	attempted_su	2022-01-02 08:00:00
r06	c001	attempted_su	2022-01-02 08:00:00

rider\_id: varchar  
city\_id: varchar  
event\_name: varchar  
timestamp: datetime

- **rider\_id** identifies the driver who signed up for the service. According to the question description, it corresponds with the **driver\_id** value from the **trip\_details** table. We'll use it to JOIN two tables.
- Values in the **event\_name** column will allow us to identify successful signups.
- **timestamp** values describe the time and date of the signup.

The **trip\_details** table contains information about trips. Let's look at important values in this table:

id	client_id	driver_id	city_id	client_rating	driver_rating	request_at	predicted_eta	actual_time_of_arrival
t01	cl12	r01	c001	4.9	4.5	2022-01-02 09:00:00	2022-01-02 09:10:00	2022-01-02 09:10:00
t02	cl10	r01	c001	4.9	4.8	2022-01-02 11:00:00	2022-01-02 11:10:00	2022-01-02 11:10:00
t03	cl9	r04	c001	4.9	4.8	2022-01-03 11:00:00	2022-01-03 11:10:00	2022-01-03 11:10:00

```

id:                varchar
client_id:         varchar
driver_id:         varchar
city_id:           varchar
client_rating:     float
driver_rating:     float
request_at:        datetime
predicted_eta:     datetime
actual_time_of_arrival: datetime
status:           varchar

```

- **driver\_id** identifies the Uber driver.
- **actual\_time\_of\_arrival** specifies the time when the trip was complete.
- We'll need to look at the **status** value to identify successful trips.

## Logical Approach

To solve this question, we need to write two queries - one that returns all drivers who signed up in the first 7 days of 2022 and another that returns all drivers who completed a trip within 168 hours of signing up. Both of these queries are quite complex, so it's best to save them as CTEs.

First, we create a query that returns all successful signups (all columns) in the first 7 days.

We filter the initial **signup\_events** table to return only successful signups. We do that by using the **LIKE** operator and specifying the 'su\_success' string.

Also, we use the AND and BETWEEN logical operators to specify the 'acceptable' range of dates: from '2022-01-01' to '2022-01-07'. In other words, the first 7 days of 2022. Then we filter signup records by date. We use the **DATE()** function to get date values from the timestamp.

Next, we create a list of drivers who completed the trip within 168 hours of signing up.

Information about signup events and completed trips are stored in different tables, so first, we need to JOIN two tables. The question description tells us that the shared dimension between the two tables is **rider\_id** and **driver\_id**.

Then we need to filter the result of JOIN to meet these two conditions:

1. Trip must be completed
2. The time difference between **time\_of\_arrival** and the signup event is under 168 hours.

Since both conditions need to be met, we should use the AND logical operator to chain these two conditions and the WHERE clause to remove rows that don't meet both conditions.

We can use the LIKE operator to make sure values in the **status** column match with the 'completed' string.

To find the difference and set a condition, simply subtract the time of the signup event (**timestamp** column) from the **actual\_time\_of\_arrival**. However, this will not give us meaningful value. We need a difference in hours, so we must use the **EXTRACT()** function.

The answer would be more accurate if we get the difference in seconds first and divide the number of seconds by 3600 (the number of seconds in an hour). The first argument to the **EXTRACT()** function is the unit of time you want to extract. To get seconds, pass EPOCH.

We should use a comparison operator to return records where the result of division (seconds by 3600) is 168 or less.

If one driver completed multiple trips within 168 hours of signing up, he should still count as one driver. To achieve this, you should use a **DISTINCT** clause when you SELECT only unique **driver\_id** values from the result of a JOIN.

Now we have a list of drivers who signed up in the first 7 days of 2022 and a list of drivers who completed a trip within 168 hours of signing up.

To answer this question, we need to find the ratio (and percentage) of drivers who completed a trip within 168 hours vs all drivers who signed up in the first 7 days.

We should use LEFT JOIN to retain all records of drivers who signed up in the first 7 days and partially fill in the information for drivers who did complete the trip in 168 hours. Then we can use COUNT() to get the number of all drivers who signed up in the first 7 days.

We also need another COUNT() to get the number of drivers who made their trip in the first 168 hours. Because of how LEFT JOIN works, some of these values will be NULL.

Finally, we need to divide the number of drivers who completed the trip in the first 168 hours by the larger number of drivers who signed up in the first 7 days. This will give us a ratio. Because the result is a float, we need to convert one of the results of a **COUNT()** function to a float as well.

To get the percentage, we multiply the ratio by 100.

## Write the code

### 1. Write a query that returns all signups in the first 7 days

We need to filter the **signup\_events** table to return successful signups in the first 7 days.

The table does not contain the date of the event. So we'll use the **DATE()** function to get a date value from the **timestamp** value.

```
SELECT *,
        DATE(timestamp)
FROM signup_events
WHERE event_name LIKE 'su_success'
      AND DATE(timestamp) BETWEEN '2022-01-01' AND '2022-01-07'
```

Next, we'll need to set up a WHERE statement and chain two conditions:

1. **event\_name** value matches 'su\_success' string
2. The date of the event falls between the 1st and 7th of January.

We use the BETWEEN operator to specify the date range.

The question asks us how many drivers who signed up in the first 7 days (the result of this query) completed the trip within 168 hours of signing up (the result of another query). Eventually, we'll have to JOIN these two queries, so we should save it as CTE. Make sure to give it a descriptive name.

### 2. Write a query that returns all drivers who completed a trip within 168 hours

We need to JOIN two tables to get information about signup events and completed trips in one table.

We need to use the ON clause to define a shared dimension - **rider\_id** value from the **signup\_events** table and **driver\_id** from the **trip\_details** table.

```
SELECT DISTINCT driver_id
FROM signup_events
JOIN trip_details ON rider_id = driver_id
```

```
WHERE status LIKE 'completed'
AND EXTRACT(EPOCH
            FROM actual_time_of_arrival - timestamp)/3600 <= 168
```

We chain an additional WHERE clause to set two conditions:

For each trip, the **status** value should match the 'completed' string and the time between signup event (**timestamp**) and **actual\_time\_of\_arrival** is under 168 hours.

We use the **EXTRACT()** function to get the number of seconds between two timestamps. We specify EPOCH to get the number of seconds from the difference between two values.

Divide the result of the EXTRACT() function by 3600 to get the number of hours between two events. That number should be under 168.

Finally, we use the DISTINCT clause to make sure that each driver counts only once, even if they made multiple trips within 168 hours.

Save the result of this query as a CTE.

### 3. Find the percentage of drivers

We have a query that returns all drivers who signed up in the first 7 days and all drivers who completed the trip within 168 hours of signing up.

The question asks us how many drivers from the second list also appear in the first list. We should perform a LEFT JOIN, where the first table is **signups** (CTE that returns people who signed up in the first 7 days), and the second table is **first\_trips\_in\_168\_hours** (self-explanatory).

LEFT JOIN will keep all records from the first table but only the matching values from the second table.

```
SELECT city_id, date, CAST(COUNT(driver_id) AS FLOAT) / COUNT(rider_id) *
100.0 AS percentage
FROM signups
LEFT JOIN first_trips_in_168_hours ON rider_id = driver_id
GROUP BY city_id, date
```

Finally, we should find the percentage of drivers who meet the criteria in each city and date. Group records by **city\_id** and **date** values.

To calculate the percentage, we use **COUNT()** to get the number of riders who completed a trip and divide it by the result of another **COUNT()** function, which finds the number of signups for each group.

The result should be a decimal value, so we convert the result of the **COUNT()** function to a float.

Finally, we multiply the ratio by 100 to get the percentage value and use the **AS** command to give it a descriptive name.

The complete query should look something like this:

```
WITH signups AS
  (SELECT *,
    DATE(timestamp)
  FROM signup_events
  WHERE event_name LIKE 'su_success'
    AND DATE(timestamp) BETWEEN '2022-01-01' AND '2022-01-07'),
  first_trips_in_168_hours AS
  (SELECT DISTINCT driver_id
  FROM signup_events
  JOIN trip_details ON rider_id = driver_id
  WHERE status LIKE 'completed'
    AND EXTRACT(EPOCH
      FROM actual_time_of_arrival - timestamp)/3600 <= 168)
SELECT city_id, date, CAST(COUNT(driver_id) AS FLOAT) / COUNT(rider_id) *
100.0 AS percentage
FROM signups
LEFT JOIN first_trips_in_168_hours ON rider_id = driver_id
GROUP BY city_id, date
```

Try running the query to see if it matches the expected output.

## Output

The output should include percentage values for each date and city.

city_id	date	percentage
c002	2022-01-06	100
c002	2022-01-04	50
c001	2022-01-05	100
c002	2022-01-02	100
c001	2022-01-04	100

### Question 3: Above Average But Not At The Top

“Find all people who earned more than the average in 2013 for their designation but were not amongst the top 5 earners for their job title. Use the totalpay column to calculate total earned and output the employee name(s) as the result.”

Link to the question:

<https://platform.stratascratch.com/coding/9985-above-average-but-not-at-the-top>

#### Available data:

We are given one **sf\_public\_salaries** table. It contains a lot of values, but only a few are important.



id	employee_name	jobtitle	basepay	overtimepay	otherpay	benefits	totalpay
120566	Teresa L Cavanaugh	EMT/Paramedic/Firefighter	100952.41	0	4254.88	34317.37	105207.29
72002	Ray Torres	Public Service Trainee	1121.28	0	0	185.77	1121.28
122662	Rizaldy T Tabada	Deputy Sheriff	92403.77	138.18	2903.94	33625.27	95445.89
140122	Gregory B	Firefighter	22757.5	0	0	22757.5	22757.5

```

id: int
employee_name: varchar
jobtitle: varchar
basepay: float
overtimepay: float
otherpay: float
benefits: float
totalpay: float
totalpaybenefits: float
year: int
notes: datetime
agency: varchar
status: varchar

```

- We don't need to identify employees, we only output their names. So the **id** value can be ignored.
- The output includes employee names, so **employee\_name** values are necessary.
- We need to find the average salary for each **jobtitle** value.
- The **totalpay** column contains salary numbers.
- Other columns - base pay and benefits can be safely ignored.
- We'll need to filter records by **year**.
- **notes**, **agency**, and **status** are not important.

## Logical Approach

Answer to this question can be separated into two parts:

First, we write a query that finds the average pay for each job title in 2013. We'll have to group records by **jobtitle** value and use the **AVG()** function to find the average salary for each job title. Then chain an additional WHERE and set a condition - **year** should be equal to 2013.

To find employees with above-average salaries, we INNER JOIN this query with the main table on **jobtitle**, the shared dimension. We need INNER JOIN to remove rows that do not meet the

criteria. We'll use the ON clause to set the condition - employees must have above-average salaries.

We need INNER JOIN because it eliminates all rows that don't meet the criteria.

To output the final answer, we need to filter records by one more condition - employees should not be among the TOP 5 earners for their job title. We'll use window functions to rank employees and filter out the top 5 employees.

Then we should set an additional WHERE clause to filter the result of INNER JOIN. We should use the IN operator to record only the employee names that appear in the list of employees with a rank of 5 or below.

## Write the code

### 1. Find average pay for job titles

The question asks us to return employees with above-average salaries. To do this, first, we need to find the average salary for each **jobtitle** in the year 2013.

```
SELECT jobtitle,  
       AVG(totalpay) AS avg_pay  
FROM sf_public_salaries  
WHERE YEAR = 2013  
GROUP BY jobtitle
```

We should group records by **jobtitle** value, and apply the **AVG()** aggregate function to **totalpay** column. Use the **AS** command to save the result of the **AVG()** function.

### 2. Filter out employees with below-average salaries

Next, we need to filter employees by their **totalpay** value.

We'll use the **avg\_pay** value as a minimum threshold. So we need to JOIN the main table with the result of the subquery that returns **avg\_pay**.

In this case, we use the ON clause (instead of WHERE) to kill two birds with one stone: JOIN the main query and subquery on a shared dimension (**jobtitle**) and discard records of employees whose **totalpay** is below the **avg\_pay** number.

```
SELECT main.employeename  
FROM sf_public_salaries main  
JOIN  
  (SELECT jobtitle,
```

```

        AVG(totalpay) AS avg_pay
    FROM sf_public_salaries
    WHERE YEAR = 2013
    GROUP BY jobtitle) aves ON main.jobtitle = aves.jobtitle
    AND main.totalpay > aves.avg_pay

```

Running this query will return all employees whose salary is above average for their position. However, the question tells us to exclude employees with TOP 5 salaries.

### 3. Get the list of employees who do not have the 5 highest salaries

We need a subquery that returns all employees, except those with the 5 highest salaries for their job title.

To accomplish this, first, we write a subquery that returns **employeeename**, **jobtitle**, and **totalpay** values of employees. Also, we use the **RANK()** window function to rank employees by their salary (compared to others with the same **jobtitle**) and assign them a number to represent their rank.

Then we should SELECT **employeeename** values of employee records where the rank value (named **rk**) is above 5.

```

SELECT employeeename
FROM
    (SELECT employeeename,
        jobtitle,
        totalpay,
        RANK() OVER (PARTITION BY jobtitle
                        ORDER BY totalpay DESC) rk
    FROM sf_public_salaries
    WHERE YEAR = 2013 ) sq
WHERE rk > 5

```

This subquery returns all employees who are NOT among TOP 5 earners for their job title.

### 4. Filter out the TOP 5 employees

In step 2, we had records of all employees with above-average salaries. The question tells us to exclude the Top 5 earners.

In step 3, we wrote a subquery that returns all employees except the TOP 5 earners.

So we need to set the WHERE clause to filter the initial list of employees with above-average salaries. We use the IN operator to return only the names of employees who:

1. Above average salary for their job title (filtered in the first JOIN)
2. Appear in the list of employees who are NOT top 5 earners for their position.

**Final code:**

```
SELECT main.employeename
FROM sf_public_salaries main
JOIN
  (SELECT jobtitle,
    AVG(totalpay) AS avg_pay
  FROM sf_public_salaries
  WHERE YEAR = 2013
  GROUP BY jobtitle) aves ON main.jobtitle = aves.jobtitle
AND main.totalpay > aves.avg_pay
WHERE main.employeename IN
  (SELECT employeename
  FROM
    (SELECT employeename,
      jobtitle,
      totalpay,
      RANK() OVER (PARTITION BY jobtitle
        ORDER BY totalpay DESC) rk
    FROM sf_public_salaries
    WHERE YEAR = 2013 ) sq
  WHERE rk > 5)
```

Run the query to see if it returns the right answer.

### Output

The query should return the name of the employee who meets both conditions - has above average salary, but not TOP 5 salary.

employeename

Michaela T Womack

## Question 4: Lyft Driver Salary and Service Tenure

“Find the correlation between the annual salary and the length of the service period of a Lyft driver.”

Link to the question:

<https://platform.stratascratch.com/coding/10018-lyft-driver-salary-and-service-tenure>

### Available data:

We are given only one table with four columns. Let's take a look at four columns:

index	start_date	end_date	yearly_salary
0	2018-04-02		48303
1	2018-05-30		67973
2	2015-04-05		56685
3	2015-01-08		51320
4	2017-03-09		67507

```
index:      int
start_date: datetime
end_date:   datetime
yearly_salary: int
```

- We won't need **index** values, which are simply incremental numbers.
- To calculate the driver's tenure, find the difference between **start\_date** and **end\_date** values
- We'll use **the yearly\_salary** value to find the correlation.

### Logical Approach

To find the correlation, we need both **yearly\_salary** and tenure values. We already have one and can find another by calculating the difference between **start\_date** and **end\_date** values.

The difficulty is that drivers still working for Lyft do not have an **end\_date** value.

We can use the **COALESCE()** function to solve this problem. It returns the first non-NULL value. We can pass it two arguments - **end\_date** and **CURRENT\_DATE**, and subtract **start\_date** from its result.

If there is a non-NULL **end\_date** value, then **COALESCE()** will return **end\_date**. If it's NULL, **COALESCE()** should return the current date, and we'll find the difference between it and **start\_date**. We should use the AS command to describe the result of this operation.

Once we have both values, we can use the **CORR()** function to calculate correlation between tenure and salary.

Finally, we should SELECT tenure and yearly salary values and pass them as arguments to **CORR()** aggregate function. **CORR()** accepts numeric values, so when we generate tenure, we should cast it to a NUMERIC type.

## Write the code

### 1. Calculate driver tenure

The table contains information about the driver's **start\_date** and **end\_date**, but not their tenure.

We calculate the tenure by finding the difference between **end\_date** and **start\_date**.

If the **end\_date** is NULL (driver is currently driving for Lyft) then **COALESCE()** function will return today's date (**CURRENT\_DATE**). We will calculate the tenure by finding the difference between today's date and **start\_date**.

To make things easier, we should also SELECT **yearly\_salary** and rename it to **salary**.

```
SELECT (COALESCE(end_date::DATE, CURRENT_DATE) - start_date::DATE)::NUMERIC
AS duration,
       yearly_salary AS salary
FROM lyft_drivers
```

This subquery should return two values - driver's tenure and their salary. We should save the result of this query as a CTE.

In the next step, we'll find a correlation between salary and tenure. We can't find a correlation between two values of different types, so we should cast **tenure** to a NUMERIC type.

### 2. Find the correlation between tenure and salary

At this point, we have a CTE that returns both **tenure** and **salary** values.

We only need to use the **CORR()** aggregate function to find a correlation between these two numbers. Correlation is going to be a decimal value, so we should cast it to a **float** type.

```
WITH df1 AS
  (SELECT (COALESCE(end_date::DATE, CURRENT_DATE) -
start_date::DATE)::NUMERIC AS duration,
         yearly_salary AS salary
   FROM lyft_drivers)
```

```
SELECT CORR(duration, salary)::float
FROM df1
```

Run the query to see if it returns the right answer.

### Output

The query should return only one value - a small decimal number to represent the correlation.

corr

0.005

## Question 5: Top 3 Wineries In The World

“Find the top 3 wineries in each country based on the average points earned. In case there is a tie, order the wineries by winery name in ascending order. Output the country along with the best, second best, and third best wineries. If there is no second winery (NULL value) output 'No second winery' and if there is no third winery output 'No third winery'. For outputting wineries format them like this: "winery (avg\_points)"""

Link to the question:

<https://platform.stratascratch.com/coding/10042-top-3-wineries-in-the-world>

### Available data:

All the necessary information is contained in one table.

id	country	description	designation	points	price	province	region_1
126576	US	Rich and round, this offers plenty of concentrated blackberry notes enveloped in warm spices and supple oak. There's a hint of green tomato leaves throughout, but the lush fruit combined with sturdy grape tannins and high	Estate Club	87	32	Virginia	Virginia

id:	int
country:	varchar
description:	varchar
designation:	varchar
points:	int
price:	float
province:	varchar
region_1:	varchar
region_2:	varchar
variety:	varchar
winery:	varchar

- **id** value identifies each winery. It can be safely ignored.
- We need to find top wineries in each country, so **country** values are important.
- We also need to find average **points** for each winery.
- Our output should include winery names from the **winery** column.

## Logical Approach

The first step should be calculating average points for wineries in each country. To achieve this, we need to group records by **country** and **winery** values and use the **AVG()** function to find average points for each group.

To avoid complexity, you should save the query as a CTE.

Next, you should rank wineries in each country by their points. You can use either ranking window functions like **RANK()**, or simply **ROW\_NUMBER()**. If you order wineries in descending order (from highest points to lowest), you can use **ROW\_NUMBER()** to assign numbers. So, the highest-scoring winery will be assigned number 1, the second-highest number 2, and so on.



The question tells us to order wineries by their names if average points are tied. The description also specifies the output format. It should have four columns: **country** and names\* of corresponding first, second, and third-place wineries.

One important detail is that winery names should be followed by corresponding average points in parentheses (Madison Ridge (84) ).

The query to number rows can be quite complex. Better save it as CTE and reference its result.

Take a look at the expected output:

country	top_winery	second_winery	third_winery
Argentina	Bodega Noemaa de Patagonia (89)	Bodega Norton (86)	Rutini (86)
Australia	Madison Ridge (84)	No second winery	No third winery
Austria	Schloss Gobelsburg (93)	Hopler (83)	No third winery
Bulgaria	Targovishte (84)	No second winery	No third winery
Chile	Altaa,r (85)	Francois Lurton (85)	Santa Carolina (85)

To achieve this, we need to use CASE/WHEN syntax in the SELECT statement to generate custom values for three columns. We'll use the AS command to give three values descriptive names - **top\_winery**, **second\_winery**, and **third\_winery**.

The CASE expression will look at **position** values to return values. If **position** is 1, then the CASE expression returns the respective name of that winery and concatenates three additional strings - an opening parenthesis, rounded average point value, and a closing parenthesis.

In SQL, || is the concatenation operator.

Finally, we group records by **country** and use the **MAX()** aggregate function to return the highest, second-highest, and third-highest values in each group.

In some countries, there is only one winery so that that country won't have second-place or third-place wineries. In this case, the question tells us to return 'No second winery' or 'No third winery'.

We can use the **COALESCE()** function to handle this edge case. As you remember, it returns the first non-NULL value. We use the **MAX()** function to find the highest **second\_place** value for

each country. If that returns NULL, then **COALESCE()** will return specified strings - 'No second winery' for the second place or 'No third winery' for the third place.

## Write the code

### 1. Average points for each winery

Each record in the **winemag\_p1** table contains wide-ranging information.

Before we rank wineries, we need to calculate the average **points** for each **winery** in each **country**.

So we **SELECT** these three values and write a **GROUP BY** statement to group records by **country** and **winery**.

In the **SELECT** statement, we use the **AVG()** aggregate function to find average points for each group.

```
SELECT country,
       winery,
       AVG(points) AS avg_points
FROM winemag_p1
WHERE country IS NOT NULL
GROUP BY country,
       winery
```

### 2. Rank wineries by their average points in each country

Next, we **SELECT** **country**, **winery**, and **avg\_points** values from the subquery result and use the **ROW\_NUMBER()** to rank wineries nationwide.

In this case, we order wineries within each **country** (partition) in descending order by their average points. Wineries with higher **avg\_points** positioned first and followed by wineries with lower **avg\_points**. We also specify that wineries with the same orders should be returned alphabetically.

**ROW\_NUMBER()** assigns number values based on the records' position in the partition. Because wineries are ranked by average points, **ROW\_NUMBER()** effectively generates a number to represent the rank of each winery. We use the **AS** command to save its result as **POSITION**.

```
SELECT country,
       winery,
```

```

ROW_NUMBER() OVER (PARTITION BY country
                    ORDER BY avg_points DESC, winery ASC) AS
POSITION,
                    avg_points
FROM
  (SELECT country,
          winery,
          AVG(points) AS avg_points
   FROM winemag_p1
   WHERE country IS NOT NULL
   GROUP BY country,
            winery) tmp1

```

### 3. Generate top\_winery, second\_winery, and third\_winery values

Next, we SELECT **country** values and use the **CASE** expression to create three values.

If the **POSITION** of a certain **winery** is 1, then **CASE** returns the name of the winery and concatenates average points placed between parenthesis.

The CASE expression that checks for wineries with a **POSITION** of 1 is named **top\_winery**. Wineries with a **POSITION** value of 2 are named **second winery**, and **POSITION** value of 3 - **third\_winery**.

If there are no wineries by positions 1, 2, or 3, our CASE expression returns NULL.

In SQL, we use the || operator to concatenate.

```

SELECT country,
       CASE
         WHEN POSITION = 1 THEN winery || ' (' || ROUND(avg_points) || ')'
         ELSE NULL
       END AS top_winery,
       CASE
         WHEN POSITION = 2 THEN winery || ' (' || ROUND(avg_points) || ')'
         ELSE NULL
       END AS second_winery,
       CASE
         WHEN POSITION = 3 THEN winery || ' (' || ROUND(avg_points) || ')'
         ELSE NULL
       END AS third_winery
FROM
  (SELECT country,
          winery,
          ROW_NUMBER() OVER (PARTITION BY country

```

```

ORDER BY avg_points DESC, winery ASC) AS POSITION,
avg_points
FROM
  (SELECT country,
         winery,
         AVG(points) AS avg_points
   FROM winemag_p1
  WHERE country IS NOT NULL
  GROUP BY country,
         winery) tmp1) tmp2
WHERE POSITION <= 3

```

We are not interested in wineries outside of TOP 3, so we set the **WHERE** clause to remove them.

#### 4. Return the highest values or a placeholder

The question asks us to return the highest values by country. We should group records by **country** and use the **MAX()** aggregate function to find the highest **top\_winery** score.

Next, we use the **COALESCE()** function to return the highest **secondary\_winery** value. If a certain country does not have the second highest **secondary\_winery** value, then **COALESCE()** returns a placeholder - 'No second winery'.

Similarly, we use **COALESCE()** to return the highest **third\_winery** value or a placeholder.

```

SELECT country,
       MAX(top_winery) AS top_winery,
       COALESCE(MAX(second_winery), 'No second winery') AS second_winery,
       COALESCE(MAX(third_winery), 'No third winery') AS third_winery
FROM
  (SELECT country,
         CASE
           WHEN POSITION = 1 THEN winery || ' (' || ROUND(avg_points) || ')'
           ELSE NULL
         END AS top_winery,
         CASE
           WHEN POSITION = 2 THEN winery || ' (' || ROUND(avg_points) || ')'
           ELSE NULL
         END AS second_winery,
         CASE
           WHEN POSITION = 3 THEN winery || ' (' || ROUND(avg_points) || ')'
           ELSE NULL
         END AS third_winery
   FROM
     (SELECT country,

```

```

        winery,
        ROW_NUMBER() OVER (PARTITION BY country
                           ORDER BY avg_points DESC, winery ASC) AS POSITION,
        avg_points
FROM
    (SELECT country,
            winery,
            AVG(points) AS avg_points
    FROM winemag_p1
    WHERE country IS NOT NULL
    GROUP BY country,
            winery) tmp1) tmp2
WHERE POSITION <= 3) tmp3
GROUP BY country

```

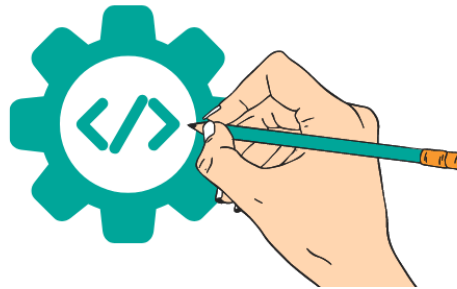
Run the query to see its result.

### Output

The query should return three values: **country**, **top\_winery**, **second\_winery**, and **third\_winery**. Not all countries will have second and third wineries. In that case, these columns should contain a placeholder.

country	top_winery	second_winery	third_winery
Argentina	Bodega Noemaa de Patagonia (89)	Bodega Norton (86)	Rutini (86)
Australia	Madison Ridge (84)	No second winery	No third winery
Austria	Schloss Gobelsburg (93)	Hopler (83)	No third winery
Bulgaria	Targovishte (84)	No second winery	No third winery
Chile	Altaar (85)	Francois Lurton (85)	Santa Carolina (85)

# Ten (Other) Strategies to Write Efficient and Advanced SQL Queries



In a professional environment, you need to write queries that are both correct and efficient. Companies often maintain millions of rows of data. Inefficient queries could lead to problems.

Let's discuss 10 additional strategies for writing efficient queries.

## 1. **SELECT column, not SELECT \***

The SELECT statement should pull up essential values, nothing more. Don't use an asterisk (\*), which SELECTs all columns.

Queries always run faster on smaller volumes of data. Specifying columns also reduces the load on the database.

## 2. **Use DISTINCT with caution**

In SQL, deduplication is resource-expensive but often a necessary task. If you use **DISTINCT**, make sure it's really necessary.

Also, use **DISTINCT** on values that uniquely identify the record. For example, if you want to remove duplicate records that describe the same person, apply **DISTINCT** to primary keys, not the **name** column. If you do make this mistake, **DISTINCT** might remove two different people with the same name.

## 3. **Don't use WHERE to combine tables**

We can use the WHERE clause to combine tables conditionally. SELECT values from two tables, and set a WHERE clause to ensure these values are equal. Instead, use JOINS and the ON clause to combine tables on a shared dimension.

JOINS are specifically designed for combining tables in SQL. Not only are JOINS more efficient, but they are also more versatile than a simple WHERE statement.

## 4. **Appropriate use of WHERE and HAVING**

Both WHERE and HAVING clauses allow you to filter data in SQL. The big difference is that the WHERE clause filters records, whereas HAVING filters the results of aggregate functions.

In SQL, aggregation is an expensive operation. Your goal is to minimize the input to aggregate functions. If you need to filter records by certain criteria, use the WHERE clause to remove rows that do not meet the criteria, then group them and aggregate values in smaller groups.

This is more efficient than the alternative - group records in the entire table, apply aggregate functions to all rows and use HAVING to filter the results.

#### **5. LIMIT to sample query results, look what your query outputs**

Use LIMIT to see a reduced number of records from the output of the query. Looking at a limited sample lets you know if you need to change the query. More importantly, it puts less pressure on the production database.

#### **6. Remove unnecessary ORDER BY**

Ordering records is a resource-expensive operation. If you're going to use ORDER BY, make sure it's necessary.

Oftentimes ORDER BY is unnecessarily used in subqueries. If the main query uses ORDER BY, there's no need to order rows in the subquery.

#### **7. Reduce the cost of JOIN**

You should always aim to reduce the size of tables before you JOIN them. Explore the possibility of removing duplicates or grouping records to reduce the number of queries in a table before you perform a JOIN.

#### **8. Correct use of wildcards**

Wildcards can be a useful tool for filtering text values. However, using wildcards to search the contents of the string can be expensive. If possible, try to use them only to evaluate the end of strings.

#### **9. GROUP BY multiple columns**

Sometimes, you must group records by two, three, or even more values. When you do, use the column with more distinct values first (id) and then the column with more common values (first name).

#### **10. UNION vs UNION ALL**

As we already mentioned, removing duplicates is an expensive operation. UNION vertically combines two datasets and takes the extra step of removing duplicate rows.

UNION ALL, on the other hand, combines the results of the queries and stops there. It is a much less expensive operation, so if your datasets do not contain duplicates or if duplicates are acceptable, UNION ALL is a more efficient choice.

## Summary

In this article, we tried to explain advanced SQL concepts and their possible use cases. We also tried to demonstrate how to use them to answer [Advanced SQL Questions](#) modeled on data scientists' actual day-to-day tasks.

Learning beyond basic SQL concepts allows you to choose the right feature (or combination of features) to write efficient and advanced SQL queries.

If you'd like to continue practicing advanced SQL concepts and features, sign up on the StrataScratch platform. Once there, you can explore hundreds of SQL interview questions of various difficulties, try to answer them, and read and participate in discussions about which SQL feature is best suited for the task.



# SQL NOTES

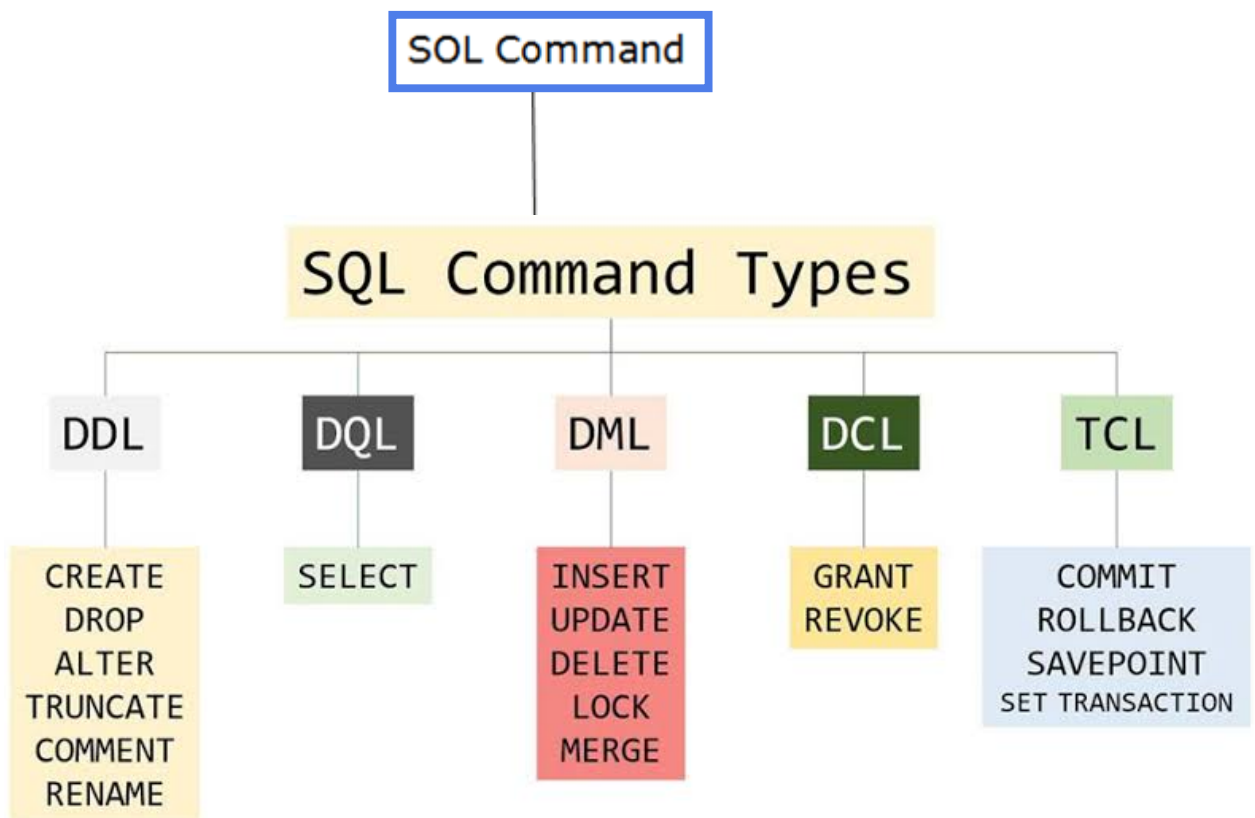
## SQL Commands

CREATE BY - ATUL KUMAR (LINKEDIN)

- SQL commands are instructions. It is used to communicate with the database. It is also used to perform specific tasks, functions, and queries of data.
- SQL can perform various tasks like create a table, add data to tables, drop the table, modify the table, set permission for users.

### Types of SQL Commands

There are five types of SQL commands: DDL, DML, DCL, TCL, and DQL.



### 1. Data Definition Language (DDL)

- DDL changes the structure of the table like creating a table, deleting a table, altering a table, etc.
- All the command of DDL are auto-committed that means it permanently save all the changes in the database.

Here are some commands that come under DDL:

- CREATE
- ALTER
- DROP
- TRUNCATE

a. **CREATE** It is used to create a new table in the database.

**Syntax:**

1. CREATE TABLE TABLE\_NAME (COLUMN\_NAME DATATYPES[,....]);

**Example:**

1. CREATE TABLE EMPLOYEE(Name VARCHAR2(20), Email VARCHAR2(100), DOB DATE);

b. **DROP**: It is used to delete both the structure and record stored in the table.

**Syntax**

1. DROP TABLE ;

**Example**

1. DROP TABLE EMPLOYEE;

c. **ALTER**: It is used to alter the structure of the database. This change could be either to modify the characteristics of an existing attribute or probably to add a new attribute.

**Syntax:**

To add a new column in the table

1. ALTER TABLE table\_name ADD column\_name COLUMN-definition;

To modify existing column in the table:

1. ALTER TABLE MODIFY(COLUMN DEFINITION....);

**EXAMPLE**

1. ALTER TABLE STU\_DETAILS ADD(ADDRESS VARCHAR2(20));
2. ALTER TABLE STU\_DETAILS MODIFY (NAME VARCHAR2(20));

d. **TRUNCATE**: It is used to delete all the rows from the table and free the space containing the table.

**Syntax:**

1. TRUNCATE TABLE table\_name;

**Example:**

1. TRUNCATE TABLE EMPLOYEE;

## 2. Data Manipulation Language

- DML commands are used to modify the database. It is responsible for all form of changes in the database.
- The command of DML is not auto-committed that means it can't permanently save all the changes in the database. They can be rollback.

Here are some commands that come under DML:

- INSERT
- UPDATE
- DELETE

a. **INSERT:** The INSERT statement is a SQL query. It is used to insert data into the row of a table.

**Syntax:**

1. INSERT INTO TABLE\_NAME
2. (col1, col2, col3,.... col N)
3. VALUES (value1, value2, value3, .... valueN);

Or

1. INSERT INTO TABLE\_NAME
2. VALUES (value1, value2, value3, .... valueN);

**For example:**

1. INSERT INTO javatpoint (Author, Subject) VALUES ("Sonoo", "DBMS");

b. **UPDATE:** This command is used to update or modify the value of a column in the table.

**Syntax:**

1. UPDATE table\_name SET [column\_name1= value1,...column\_nameN = valueN] [WHERE CONDITION]

**For example:**

1. UPDATE students
2. SET User\_Name = 'Sonoo'
3. WHERE Student\_Id = '3'

c. **DELETE:** It is used to remove one or more row from a table.

**Syntax:**

1. DELETE FROM table\_name [WHERE condition];

**For example:**

1. DELETE FROM javatpoint
2. WHERE Author="Sonoo";

### 3. Data Control Language

DCL commands are used to grant and take back authority from any database user.

Here are some commands that come under DCL:

- Grant
- Revoke

a. **Grant:** It is used to give user access privileges to a database.

**Example**

1. GRANT SELECT, UPDATE ON MY\_TABLE TO SOME\_USER, ANOTHER\_USER;

b. **Revoke:** It is used to take back permissions from the user.

**Example**

1. REVOKE SELECT, UPDATE ON MY\_TABLE FROM USER1, USER2;

### 4. Transaction Control Language

TCL commands can only use with DML commands like INSERT, DELETE and UPDATE only.

These operations are automatically committed in the database that's why they cannot be used while creating tables or dropping them.

Here are some commands that come under TCL:

- COMMIT
- ROLLBACK
- SAVEPOINT

a. **Commit:** Commit command is used to save all the transactions to the database.

**Syntax:**

1. COMMIT;

**Example:**

1. DELETE FROM CUSTOMERS
2. WHERE AGE = 25;
3. COMMIT;

b. **Rollback:** Rollback command is used to undo transactions that have not already been saved to the database.

**Syntax:**

1. ROLLBACK;

**Example:**

1. DELETE FROM CUSTOMERS
2. WHERE AGE = 25;
3. ROLLBACK;

c. **SAVEPOINT:** It is used to roll the transaction back to a certain point without rolling back the entire transaction.

**Syntax:**

1. SAVEPOINT SAVEPOINT\_NAME;

## 5. Data Query Language

DQL is used to fetch the data from the database.

It uses only one command:

- SELECT

a. **SELECT:** This is the same as the projection operation of relational algebra. It is used to select the attribute based on the condition described by WHERE clause.

**Syntax:**

1. SELECT expressions
2. FROM TABLES
3. WHERE conditions;

**For example:**

1. SELECT emp\_name
2. FROM employee
3. WHERE age > 20;

- **CREATE TABLE**

The SQL **CREATE TABLE** statement is used to create a new table.

**Syntax**

The basic syntax of the CREATE TABLE statement is as follows –

```
CREATE TABLE table_name(  
    column1 datatype,  
    column2 datatype,  
    column3 datatype,  
    .....  
    columnN datatype,  
    PRIMARY KEY( one or more columns )  
);
```

```
CREATE TABLE Employees_details(  
    ID int,  
    Name varchar(20),  
    Address varchar(20)  
);
```

Ex:

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

- **DROP TABLE**

The SQL **DROP TABLE** statement is used to remove a table definition and all the data, indexes, triggers, constraints and permission specifications for that table.

**NOTE** – You should be very careful while using this command because once a table is deleted then all the information available in that table will also be lost forever.

#### Syntax

The basic syntax of this DROP TABLE statement is as follows –

DROP TABLE table\_name;

- INSERT INTO**

The SQL **INSERT INTO** Statement is used to add new rows of data to a table in the database.

#### Syntax

There are two basic syntaxes of the INSERT INTO statement which are shown below.

```
INSERT INTO TABLE_NAME (column1, column2, column3,...columnN)
VALUES (value1, value2, value3,...valueN);
```

Here, column1, column2, column3,...columnN are the names of the columns in the table into which you want to insert the data.

You may not need to specify the column(s) name in the SQL query if you are adding values for all the columns of the table. But make sure the order of the values is in the same order as the columns in the table.

The **SQL INSERT INTO** syntax will be as follows –

```
INSERT INTO TABLE_NAME VALUES (value1,value2,value3,...valueN);
```

#### Example

The following statements would create six records in the CUSTOMERS table.

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );
```

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );
```

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );
```

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );
```

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );
```

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6, 'Komal', 22, 'MP', 4500.00 );
```

You can create a record in the CUSTOMERS table by using the second syntax as shown below.

```
INSERT INTO CUSTOMERS
VALUES (7, 'Muffy', 24, 'Indore', 10000.00 );
```

All the above statements would produce the following records in the CUSTOMERS table as shown below.

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

- The Select Query**

The SQL **SELECT** statement is used to fetch the data from a database table which returns this data in the form of a result table. These result tables are called result-sets.

Syntax:

The basic syntax of the SELECT statement is as follows –

SELECT column1, column2, columnN FROM table\_name;

Here, column1, column2... are the fields of a table whose values you want to fetch. If you want to fetch all the fields available in the field, then you can use the following syntax.

```
SELECT * FROM table_name;
```

Example:

Consider the CUSTOMERS table having the following records –

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

The following code is an example, which would fetch the ID, Name and Salary fields of the customers available in CUSTOMERS table.

```
SQL> SELECT ID, NAME, SALARY FROM CUSTOMERS;
```

This would produce the following result –

ID	NAME	SALARY
1	Ramesh	2000.00
2	Khilan	1500.00



3	kaushik	2000.00
4	Chaitali	6500.00
5	Hardik	8500.00
6	Komal	4500.00
7	Muffy	10000.00

If you want to fetch all the fields of the CUSTOMERS table, then you should use the following query.

```
SQL> SELECT * FROM CUSTOMERS;
```

This would produce the result as shown below.

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

### • WHERE Clause

The SQL **WHERE** clause is used to specify a condition while fetching the data from a single table or by joining with multiple tables. If the given condition is satisfied, then only it returns a specific value from the table. You should use the WHERE clause to filter the records and fetching only the necessary records.

The WHERE clause is not only used in the SELECT statement, but it is also used in the UPDATE, DELETE statement, etc., which we would examine in the subsequent chapters.

### Syntax

The basic syntax of the SELECT statement with the WHERE clause is as shown below.

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition]
```

You can specify a condition using the comparison or logical operators like >, <, =, **LIKE**, **NOT**, etc. The following examples would make this concept clear.

### Example

Consider the CUSTOMERS table having the following records –

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

-----+

The following code is an example which would fetch the ID, Name and Salary fields from the CUSTOMERS table, where the salary is greater than 2000 –

```
SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY > 2000;
```

This would produce the following result –

```
-----+
| ID | NAME   | SALARY |
-----+
| 4 | Chaitali | 6500.00 |
| 5 | Hardik  | 8500.00 |
| 6 | Komal   | 4500.00 |
| 7 | Muffy   | 10000.00 |
-----+
```

The following query is an example, which would fetch the ID, Name and Salary fields from the CUSTOMERS table for a customer with the name **Hardik**.

Here, it is important to note that all the strings should be given inside single quotes (''). Whereas, numeric values should be given without any quote as in the above example.

```
SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE NAME = 'Hardik';
```

This would produce the following result –

```
-----+
| ID | NAME   | SALARY |
-----+
| 5 | Hardik  | 8500.00 |
-----+
```

- **AND and OR Conjunctive Operators**

The SQL **AND** & **OR** operators are used to combine multiple conditions to narrow data in an SQL statement. These two operators are called as the conjunctive operators.

These operators provide a means to make multiple comparisons with different operators in the same SQL statement.

- The AND Operator

The **AND** operator allows the existence of multiple conditions in an SQL statement's WHERE clause.

#### Syntax

The basic syntax of the AND operator with a WHERE clause is as follows –

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition1] AND [condition2]...AND [conditionN];
```

You can combine N number of conditions using the AND operator. For an action to be taken by the SQL statement, whether it be a transaction or a query, all conditions separated by the AND must be TRUE.

## Example

Consider the CUSTOMERS table having the following records –

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

Following is an example, which would fetch the ID, Name and Salary fields from the CUSTOMERS table, where the salary is greater than 2000 and the age is less than 25 years –

```
SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY > 2000 AND age < 25;
```

This would produce the following result –

ID	NAME	SALARY
6	Komal	4500.00
7	Muffy	10000.00

### ▪ The OR Operator

The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.

## Syntax

The basic syntax of the OR operator with a WHERE clause is as follows –

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition1] OR [condition2]...OR [conditionN]
```

You can combine N number of conditions using the OR operator. For an action to be taken by the SQL statement, whether it be a transaction or query, the only any ONE of the conditions separated by the OR must be TRUE.

## Example

Consider the CUSTOMERS table having the following records –

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

The following code block has a query, which would fetch the ID, Name and Salary fields from the CUSTOMERS table, where the salary is greater than 2000 OR the age is less than 25 years.

```
SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY > 2000 OR age < 25;
```

This would produce the following result –

ID	NAME	SALARY
3	kaushik	2000.00
4	Chaitali	6500.00
5	Hardik	8500.00
6	Komal	4500.00
7	Muffy	10000.00

#### • UPDATE Query

The SQL **UPDATE** Query is used to modify the existing records in a table. You can use the WHERE clause with the UPDATE query to update the selected rows, otherwise all the rows would be affected.

#### Syntax

The basic syntax of the UPDATE query with a WHERE clause is as follows –

```
UPDATE table_name
SET column1 = value1, column2 = value2..., columnN = valueN
WHERE [condition];
```

You can combine N number of conditions using the AND or the OR operators.

#### Example

Consider the CUSTOMERS table having the following records –

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

The following query will update the ADDRESS for a customer whose ID number is 6 in the table.

```
SQL> UPDATE CUSTOMERS
SET ADDRESS = 'Pune'
WHERE ID = 6;
```

Now, the CUSTOMERS table would have the following records –

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	Pune	4500.00
7	Muffy	24	Indore	10000.00

If you want to modify all the ADDRESS and the SALARY column values in the CUSTOMERS table, you do not need to use the WHERE clause as the UPDATE query would be enough as shown in the following code block.

```
SQL> UPDATE CUSTOMERS
SET ADDRESS = 'Pune', SALARY = 1000.00;
```

Now, CUSTOMERS table would have the following records –

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Pune	1000.00
2	Khilan	25	Pune	1000.00
3	kaushik	23	Pune	1000.00
4	Chaitali	25	Pune	1000.00
5	Hardik	27	Pune	1000.00
6	Komal	22	Pune	1000.00
7	Muffy	24	Pune	1000.00

### • DELETE Query

The SQL DELETE Query is used to delete the existing records from a table.

You can use the WHERE clause with a DELETE query to delete the selected rows, otherwise all the records would be deleted.

### Syntax

The basic syntax of the DELETE query with the WHERE clause is as follows –

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

You can combine N number of conditions using AND or OR operators.

### Example

Consider the CUSTOMERS table having the following records –

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

The following code has a query, which will DELETE a customer, whose ID is 6.

```
SQL> DELETE FROM CUSTOMERS
WHERE ID = 6;
```

Now, the CUSTOMERS table would have the following records.

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
7	Muffy	24	Indore	10000.00

If you want to DELETE all the records from the CUSTOMERS table, you do not need to use the WHERE clause and the DELETE query would be as follows –

```
SQL> DELETE FROM CUSTOMERS;
```

Now, the CUSTOMERS table would not have any record.

### • LIKE Clause

The SQL **LIKE** clause is used to compare a value to similar values using wildcard operators. There are two wildcards used in conjunction with the LIKE operator.

- The percent sign (%)
- The underscore (\_)

The percent sign represents zero, one or multiple characters. The underscore represents a single number or character. These symbols can be used in combinations.

Syntax

The basic syntax of % and \_ is as follows –

```
SELECT FROM table_name
WHERE column LIKE 'XXXX%'
```

or

```
SELECT FROM table_name
WHERE column LIKE '%XXXX%'
```

or

```
SELECT FROM table_name
WHERE column LIKE 'XXXX_'
```

or

```
SELECT FROM table_name  
WHERE column LIKE '_XXXX'
```

or

```
SELECT FROM table_name  
WHERE column LIKE ' _XXXX_'
```

You can combine N number of conditions using AND or OR operators. Here, XXXX could be any numeric or string value.

Example

The following table has a few examples showing the WHERE part having different LIKE clause with '%' and '\_' operators –

Sr.No.	Statement & Description
1	<b>WHERE SALARY LIKE '200%'</b> Finds any values that start with 200.
2	<b>WHERE SALARY LIKE '%200%'</b> Finds any values that have 200 in any position.
3	<b>WHERE SALARY LIKE '_00%'</b> Finds any values that have 00 in the second and third positions.
4	<b>WHERE SALARY LIKE '2_ _%'</b> Finds any values that start with 2 and are at least 3 characters in length.
5	<b>WHERE SALARY LIKE '%2'</b> Finds any values that end with 2.
6	<b>WHERE SALARY LIKE '_2%3'</b> Finds any values that have a 2 in the second position and end with a 3.
7	<b>WHERE SALARY LIKE '2_ _ _3'</b> Finds any values in a five-digit number that start with 2 and end with 3.

Let us take a real example, consider the CUSTOMERS table having the records as shown below.

```
+---+-----+---+-----+-----+  
| 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 |
+---+-----+-----+-----+

```

Following is an example, which would display all the records from the CUSTOMERS table, where the SALARY starts with 200.

```

SQL> SELECT * FROM CUSTOMERS
WHERE SALARY LIKE '200%';

```

This would produce the following result –

```

+---+-----+-----+-----+
| ID | NAME | AGE | ADDRESS | SALARY |
+---+-----+-----+-----+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
+---+-----+-----+-----+

```

#### • WILDCARD Operator

The SQL LIKE operator, which is used to compare a value to similar values using the wildcard operators.

SQL supports two wildcard operators in conjunction with the LIKE operator which are explained in detail in the following table.

Sr.No.	Wildcard & Description
1	<p><b>The percent sign (%)</b></p> <p>Matches one or more characters.</p> <p><b>Note</b> – MS Access uses the asterisk (*) wildcard character instead of the percent sign (%) wildcard character.</p>
2	<p><b>The underscore (_)</b></p> <p>Matches one character.</p> <p><b>Note</b> – MS Access uses a question mark (?) instead of the underscore (_) to match any one character.</p>

The percent sign represents zero, one or multiple characters. The underscore represents a single number or a character. These symbols can be used in combinations.

#### Syntax

The basic syntax of a '%' and a '\_' operator is as follows.

```

SELECT * FROM table_name
WHERE column LIKE 'XXXX%'

```

or

```

SELECT * FROM table_name

```



WHERE column LIKE '%XXXX%'

or

```
SELECT * FROM table_name  
WHERE column LIKE 'XXXX_'
```

or

```
SELECT * FROM table_name  
WHERE column LIKE '_XXXX'
```

or

```
SELECT * FROM table_name  
WHERE column LIKE '_XXXX_'
```

You can combine N number of conditions using the AND or the OR operators. Here, XXXX could be any numeric or string value.

Example

The following table has a number of examples showing the WHERE part having different LIKE clauses with '%' and '\_' operators.

Sr.No.	Statement & Description
1	<b>WHERE SALARY LIKE '200%'</b> Finds any values that start with 200.
2	<b>WHERE SALARY LIKE '%200%'</b> Finds any values that have 200 in any position.
3	<b>WHERE SALARY LIKE '_00%'</b> Finds any values that have 00 in the second and third positions.
4	<b>WHERE SALARY LIKE '2_%_%'</b> Finds any values that start with 2 and are at least 3 characters in length.
5	<b>WHERE SALARY LIKE '%2'</b> Finds any values that end with 2.
6	<b>WHERE SALARY LIKE '_2%3'</b> Finds any values that have a 2 in the second position and end with a 3.

7

**WHERE SALARY LIKE '2\_\_\_3'**

Finds any values in a five-digit number that start with 2 and end with 3.

Let us take a real example, consider the CUSTOMERS table having the following records.

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

The following code block is an example, which would display all the records from the CUSTOMERS table where the SALARY starts with 200.

```
SQL> SELECT * FROM CUSTOMERS
WHERE SALARY LIKE '200%';
```

This would produce the following result.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
3	kaushik	23	Kota	2000.00

### • TOP, LIMIT or ROWNUM Clause

The SQL **TOP** clause is used to fetch a TOP N number or X percent records from a table.

**Note** – All the databases do not support the TOP clause. For example MySQL supports the **LIMIT** clause to fetch limited number of records while Oracle uses the **ROWNUM** command to fetch a limited number of records.

Syntax

The basic syntax of the TOP clause with a SELECT statement would be as follows.

```
SELECT TOP number|percent column_name(s)
FROM table_name
WHERE [condition]
```

Example

Consider the CUSTOMERS table having the following records –

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

The following query is an example on the SQL server, which would fetch the top 3 records from the CUSTOMERS table.

```
SQL> SELECT TOP 3 * FROM CUSTOMERS;
```

This would produce the following result –

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00

If you are using MySQL server, then here is an equivalent example –

```
SQL> SELECT * FROM CUSTOMERS
LIMIT 3;
```

This would produce the following result –

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00

If you are using an Oracle server, then the following code block has an equivalent example.

```
SQL> SELECT * FROM CUSTOMERS
WHERE ROWNUM <= 3;
```

This would produce the following result –

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00

### • ORDER BY Clause

The SQL **ORDER BY** clause is used to sort the data in ascending or descending order, based on one or more columns. Some databases sort the query results in an ascending order by default.

Syntax

The basic syntax of the ORDER BY clause is as follows –

```
SELECT column-list
```

FROM table\_name  
[WHERE condition]  
[ORDER BY column1, column2, .. columnN] [ASC | DESC];

You can use more than one column in the ORDER BY clause. Make sure whatever column you are using to sort that column should be in the column-list.

#### Example

Consider the CUSTOMERS table having the following records –

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

The following code block has an example, which would sort the result in an ascending order by the NAME and the SALARY –

```
SQL> SELECT * FROM CUSTOMERS  
ORDER BY NAME, SALARY;
```

This would produce the following result –

ID	NAME	AGE	ADDRESS	SALARY
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
3	kaushik	23	Kota	2000.00
2	Khilan	25	Delhi	1500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00
1	Ramesh	32	Ahmedabad	2000.00

The following code block has an example, which would sort the result in the descending order by NAME.

```
SQL> SELECT * FROM CUSTOMERS  
ORDER BY NAME DESC;
```

This would produce the following result –

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
7	Muffy	24	Indore	10000.00
6	Komal	22	MP	4500.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
5	Hardik	27	Bhopal	8500.00
4	Chaitali	25	Mumbai	6500.00

- **Group By Clause**

The SQL **GROUP BY** clause is used in collaboration with the SELECT statement to arrange identical data into groups. This GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.

#### Syntax

The basic syntax of a GROUP BY clause is shown in the following code block. The GROUP BY clause must follow the conditions in the WHERE clause and must precede the ORDER BY clause if one is used.

```
SELECT column1, column2
FROM table_name
WHERE [ conditions ]
GROUP BY column1, column2
ORDER BY column1, column2
```

#### Example

Consider the CUSTOMERS table is having the following records –

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

If you want to know the total amount of the salary on each customer, then the GROUP BY query would be as follows.

```
SQL> SELECT NAME, SUM(SALARY) FROM CUSTOMERS
      GROUP BY NAME;
```

This would produce the following result –

NAME	SUM(SALARY)
Chaitali	6500.00
Hardik	8500.00
kaushik	2000.00
Khilan	1500.00
Komal	4500.00
Muffy	10000.00
Ramesh	2000.00

Now, let us look at a table where the CUSTOMERS table has the following records with duplicate names –

ID	NAME	AGE	ADDRESS	SALARY
----	------	-----	---------	--------

1	Ramesh	32	Ahmedabad	2000.00
2	Ramesh	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	kaushik	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Now again, if you want to know the total amount of salary on each customer, then the GROUP BY query would be as follows –

```
SQL> SELECT NAME, SUM(SALARY) FROM CUSTOMERS
GROUP BY NAME;
```

This would produce the following result –

NAME	SUM(SALARY)
Hardik	8500.00
kaushik	8500.00
Komal	4500.00
Muffy	10000.00
Ramesh	3500.00

### • Distinct Keyword

The SQL **DISTINCT** keyword is used in conjunction with the SELECT statement to eliminate all the duplicate records and fetching only unique records.

There may be a situation when you have multiple duplicate records in a table. While fetching such records, it makes more sense to fetch only those unique records instead of fetching duplicate records.

### Syntax

The basic syntax of DISTINCT keyword to eliminate the duplicate records is as follows –

```
SELECT DISTINCT column1, column2,.....columnN
FROM table_name
WHERE [condition]
```

### Example

Consider the CUSTOMERS table having the following records –

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

First, let us see how the following SELECT query returns the duplicate salary records.

```
SQL> SELECT SALARY FROM CUSTOMERS  
ORDER BY SALARY;
```

This would produce the following result, where the salary (2000) is coming twice which is a duplicate record from the original table.

```
+-----+  
| SALARY |  
+-----+  
| 1500.00 |  
| 2000.00 |  
| 2000.00 |  
| 4500.00 |  
| 6500.00 |  
| 8500.00 |  
| 10000.00 |  
+-----+
```

Now, let us use the DISTINCT keyword with the above SELECT query and then see the result.

```
SQL> SELECT DISTINCT SALARY FROM CUSTOMERS  
ORDER BY SALARY;
```

This would produce the following result where we do not have any duplicate entry.

```
+-----+  
| SALARY |  
+-----+  
| 1500.00 |  
| 2000.00 |  
| 4500.00 |  
| 6500.00 |  
| 8500.00 |  
| 10000.00 |  
+-----+
```

#### • Alias query

You can rename a table or a column temporarily by giving another name known as **Alias**. The use of table aliases is to rename a table in a specific SQL statement. The renaming is a temporary change and the actual table name does not change in the database. The column aliases are used to rename a table's columns for the purpose of a particular SQL query.

#### Syntax

The basic syntax of a **table** alias is as follows.

```
SELECT column1, column2....  
FROM table_name AS alias_name  
WHERE [condition];
```

The basic syntax of a **column** alias is as follows.

```
SELECT column_name AS alias_name  
FROM table_name  
WHERE [condition];
```

#### Example

Consider the following two tables.

**Table 1** – CUSTOMERS Table is as follows.

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 is as follows.

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, the following code block shows the usage of a **table alias**.

```
SQL> SELECT C.ID, C.NAME, C.AGE, O.AMOUNT
FROM CUSTOMERS AS C, ORDERS AS O
WHERE C.ID = O.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

Following is the usage of a **column alias**.

```
SQL> SELECT ID AS CUSTOMER_ID, NAME AS CUSTOMER_NAME
FROM CUSTOMERS
WHERE SALARY IS NOT NULL;
```

This would produce the following result.

CUSTOMER_ID	CUSTOMER_NAME
1	Ramesh
2	Khilan
3	kaushik
4	Chaitali
5	Hardik
6	Komal



7	Muffy
---	-------

## • UNIONS CLAUSE

The SQL UNION clause/operator is used to combine the results of two or more SELECT statements without returning any duplicate rows.

To use this UNION clause, each SELECT statement must have

- The same number of columns selected
- The same number of column expressions
- The same data type and
- Have them in the same order

But they need not have to be in the same length.

Syntax

The basic syntax of a **UNION** clause is as follows –

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

UNION

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

Here, the given condition could be any given expression based on your requirement.

Example

Consider the following two tables.

**Table 1** – CUSTOMERS Table is as follows.

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 is as follows.

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 follows –

```
SQL> SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
LEFT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
UNION
SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
RIGHT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result –

ID	NAME	AMOUNT	DATE
1	Ramesh	NULL	NULL
2	Khilan	1560	2009-11-20 00:00:00
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00
5	Hardik	NULL	NULL
6	Komal	NULL	NULL
7	Muffy	NULL	NULL

- The UNION ALL Clause

The UNION ALL operator is used to combine the results of two SELECT statements including duplicate rows.

The same rules that apply to the UNION clause will apply to the UNION ALL operator.

#### Syntax

The basic syntax of the **UNION ALL** is as follows.

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

#### UNION ALL

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

Here, the given condition could be any given expression based on your requirement.

#### Example

Consider the following two tables,

**Table 1** – CUSTOMERS Table is as follows.

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 is as follows.

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 follows –

```
SQL> SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
LEFT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
UNION ALL
SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
RIGHT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result –

ID	NAME	AMOUNT	DATE
1	Ramesh	NULL	NULL
2	Khilan	1560	2009-11-20 00:00:00
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00
5	Hardik	NULL	NULL
6	Komal	NULL	NULL
7	Muffy	NULL	NULL
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
2	Khilan	1560	2009-11-20 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00

There are two other clauses (i.e., operators), which are like the UNION clause.

- **SQL INTERSECT Clause** – This is used to combine two SELECT statements, but returns rows only from the first SELECT statement that are identical to a row in the second SELECT statement.

- SQL EXCEPT Clause – This combines two SELECT statements and returns rows from the first SELECT statement that are not returned by the second SELECT statement.

### • Using 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.

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 –

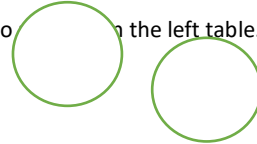
- INNER JOIN – returns rows when there is a match in both tables.



- 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.



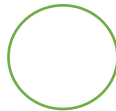
- FULL JOIN – returns rows when there is a match in one of the tables.



- 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.

- CARTESIAN JOIN – returns the Cartesian product of the sets of records from the two or more joined tables.

### 1. INNER JOIN



- The most important and frequently used of the joins is the **INNER JOIN**. They are also referred to as an **EQUIJOIN**.
- The INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate. The query compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row.
- Syntax
- The basic syntax of the **INNER JOIN** is as follows.
- SELECT table1.column1, table1.column2, table2.column2...
- FROM table1
- INNER JOIN table2
- ON table1.common\_field = table2.common\_field;
- Example
- Consider the following two tables.
- **Table 1** – CUSTOMERS Table is as follows.

	ID	NAME	AGE	ADDRESS	SALARY
1	1	Ramesh	32	Ahmedabad	2000.00
2	2	Khilan	25	Delhi	1500.00
3	3	kaushik	23	Kota	2000.00
4	4	Chaitali	25	Mumbai	6500.00
5	5	Hardik	27	Bhopal	8500.00
6	6	Komal	22	MP	4500.00
7	7	Muffy	24	Indore	10000.00

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

- **Table 2 – ORDERS** Table is as follows.

- +---+-----+-----+-----+-----+
- | 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 using the INNER JOIN as follows –

- SQL> SELECT ID, NAME, AMOUNT, DATE
- FROM CUSTOMERS
- INNER JOIN ORDERS
- ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

- This would produce the following result.

- +---+-----+-----+-----+-----+
- | ID | NAME | AMOUNT | DATE |
- +---+-----+-----+-----+-----+
- | 3 | kaushik | 3000 | 2009-10-08 00:00:00 |
- | 3 | kaushik | 1500 | 2009-10-08 00:00:00 |
- | 2 | Khilan | 1560 | 2009-11-20 00:00:00 |
- | 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |
- +---+-----+-----+-----+-----+

## 2. LEFT JOIN

The SQL **LEFT JOIN** returns all rows from the left table, even if there are no matches in the right table. This means that if the ON clause matches 0 (zero) records in the right table; the join will still return a row in the result, but with NULL in each column from the right table.

This means that a left join returns all the values from the left table, plus matched values from the right table or NULL in case of no matching join predicate.

### Syntax

The basic syntax of a **LEFT JOIN** is as follows.

```
SELECT table1.column1, table2.column2...
FROM table1
LEFT JOIN table2
ON table1.common_field = table2.common_field;
```

Here, the given condition could be any given expression based on your requirement.

### Example

Consider the following two tables,

**Table 1 – CUSTOMERS** Table is as follows.

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 is as follows.

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 using the LEFT JOIN as follows.

```
SQL> SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
LEFT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result –

ID	NAME	AMOUNT	DATE
1	Ramesh	NULL	NULL
2	Khilan	1560	2009-11-20 00:00:00
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00
5	Hardik	NULL	NULL
6	Komal	NULL	NULL
7	Muffy	NULL	NULL

### 3.RIGHT JOIN

The SQL **RIGHT JOIN** returns all rows from the right table, even if there are no matches in the left table. This means that if the ON clause matches 0 (zero) records in the left table; the join will still return a row in the result, but with NULL in each column from the left table.

This means that a right join returns all the values from the right table, plus matched values from the left table or NULL in case of no matching join predicate.

Syntax

The basic syntax of a **RIGHT JOIN** is as follow.

```
SELECT table1.column1, table2.column2...
```

```
FROM table1
RIGHT JOIN table2
ON table1.common_field = table2.common_field;
```

Example

Consider the following two tables,

**Table 1** – CUSTOMERS Table is as follows.

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 is as follows.

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 using the RIGHT JOIN as follows.

```
SQL> SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
RIGHT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result –

ID	NAME	AMOUNT	DATE
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
2	Khilan	1560	2009-11-20 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00

#### 4. FULL JOIN

The SQL **FULL JOIN** combines the results of both left and right outer joins.

The joined table will contain all records from both the tables and fill in NULLs for missing matches on either side.

Syntax



The basic syntax of a **FULL JOIN** is as follows –

```
SELECT table1.column1, table2.column2...
FROM table1
FULL JOIN table2
ON table1.common_field = table2.common_field;
```

Here, the given condition could be any given expression based on your requirement.

#### Example

Consider the following two tables.

**Table 1** – CUSTOMERS Table is as follows.

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 is as follows.

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 using FULL JOIN as follows.

```
SQL> SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
FULL JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result –

ID	NAME	AMOUNT	DATE
1	Ramesh	NULL	NULL
2	Khilan	1560	2009-11-20 00:00:00
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00
5	Hardik	NULL	NULL
6	Komal	NULL	NULL
7	Muffy	NULL	NULL
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00

```
| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |
| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |
+-----+-----+-----+
```

If your Database does not support FULL JOIN (MySQL does not support FULL JOIN), then you can use **UNION ALL** clause to combine these two JOINS as shown below.

```
SQL> SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
LEFT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
UNION ALL
SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
RIGHT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
```

## 5.SELF JOINS

The SQL **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.

Syntax

The basic syntax of SELF JOIN is as follows –

```
SELECT a.column_name, b.column_name...
FROM table1 a, table1 b
WHERE a.common_field = b.common_field;
```

Here, the WHERE clause could be any given expression based on your requirement.

Example

Consider the following table.

**CUSTOMERS Table** is as follows.

```
+-----+-----+-----+
| 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 |
+-----+-----+-----+
```

Now, let us join this table using SELF JOIN as follows –

```
SQL> SELECT a.ID, b.NAME, a.SALARY
FROM CUSTOMERS a, CUSTOMERS b
WHERE a.SALARY < b.SALARY;
```

This would produce the following result –

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

ID	NAME	SALARY
2	Ramesh	1500.00
2	kaushik	1500.00
1	Chaitali	2000.00
2	Chaitali	1500.00
3	Chaitali	2000.00
6	Chaitali	4500.00
1	Hardik	2000.00
2	Hardik	1500.00
3	Hardik	2000.00
4	Hardik	6500.00
6	Hardik	4500.00
1	Komal	2000.00
2	Komal	1500.00
3	Komal	2000.00
1	Muffy	2000.00
2	Muffy	1500.00
3	Muffy	2000.00
4	Muffy	6500.00
5	Muffy	8500.00
6	Muffy	4500.00

## 6. CARTESIAN JOIN

The CARTESIAN JOIN or CROSS JOIN returns the Cartesian product of the sets of records from two or more joined tables. Thus, it equates to an inner join where the join-condition always evaluates to either True or where the join-condition is absent from the statement.

Syntax

The basic syntax of the **CARTESIAN JOIN** or the **CROSS JOIN** is as follows –

```
SELECT table1.column1, table2.column2...
FROM table1, table2 [, table3 ]
```

Example

Consider the following two tables.

**Table 1** – CUSTOMERS table is as follows.

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 is as follows –

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 using CARTESIAN JOIN as follows –

```
SQL> SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS, ORDERS;
```

This would produce the following result –

ID	NAME	AMOUNT	DATE
1	Ramesh	3000	2009-10-08 00:00:00
1	Ramesh	1500	2009-10-08 00:00:00
1	Ramesh	1560	2009-11-20 00:00:00
1	Ramesh	2060	2008-05-20 00:00:00
2	Khilan	3000	2009-10-08 00:00:00
2	Khilan	1500	2009-10-08 00:00:00
2	Khilan	1560	2009-11-20 00:00:00
2	Khilan	2060	2008-05-20 00:00:00
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
3	kaushik	1560	2009-11-20 00:00:00
3	kaushik	2060	2008-05-20 00:00:00
4	Chaitali	3000	2009-10-08 00:00:00
4	Chaitali	1500	2009-10-08 00:00:00
4	Chaitali	1560	2009-11-20 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00
5	Hardik	3000	2009-10-08 00:00:00
5	Hardik	1500	2009-10-08 00:00:00
5	Hardik	1560	2009-11-20 00:00:00
5	Hardik	2060	2008-05-20 00:00:00
6	Komal	3000	2009-10-08 00:00:00
6	Komal	1500	2009-10-08 00:00:00
6	Komal	1560	2009-11-20 00:00:00
6	Komal	2060	2008-05-20 00:00:00
7	Muffy	3000	2009-10-08 00:00:00
7	Muffy	1500	2009-10-08 00:00:00
7	Muffy	1560	2009-11-20 00:00:00
7	Muffy	2060	2008-05-20 00:00:00

### • 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](#) chapter, but it's worth to revise them at this point.

- NOT NULL Constraint – Ensures that a column cannot have NULL value.
- DEFAULT Constraint – Provides a default value for a column when none is specified.
- UNIQUE Constraint – Ensures that all values in a column are different.
- PRIMARY Key – Uniquely identifies each row/record in a database table.
- FOREIGN Key – Uniquely identifies a row/record in any of the given database table.
- CHECK Constraint – The CHECK constraint ensures that all the values in a column satisfies certain conditions.
- INDEX – Used to create and retrieve data from the database very quickly.

Constraints can be specified when a table is created with the CREATE TABLE statement or you can use the ALTER TABLE statement to create constraints even after the table is created.

1. **NOT NULL :** By default, a column can hold NULL values. If you do not want a column to have a NULL value, then you need to define such a constraint on this column specifying that NULL is now not allowed for that column.

A NULL is not the same as no data, rather, it represents unknown data.

#### Example

For example, the following SQL query creates a new table called CUSTOMERS and adds five columns, three of which, are ID NAME and AGE, In this we specify not to accept NULLs –

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

If CUSTOMERS 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 CUSTOMERS
MODIFY SALARY DECIMAL (18, 2) NOT NULL;
```

2. **DEFAULT CONSTRAINT:** The DEFAULT constraint provides a default value to a column when the INSERT INTO statement does not provide a specific value.

#### Example

For example, the following SQL creates a new table called CUSTOMERS and adds five columns. Here, the SALARY column is set to 5000.00 by default, so in case the INSERT INTO statement does not provide a value for this column, then by default this column would be set to 5000.00.

```
CREATE TABLE CUSTOMERS(
  ID INT      NOT NULL,
  NAME VARCHAR (20)  NOT NULL,
  AGE INT      NOT NULL,
  ADDRESS CHAR (25) ,
```

```
SALARY DECIMAL (18, 2) DEFAULT 5000.00,  
PRIMARY KEY (ID)  
);
```

If the CUSTOMERS 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 CUSTOMERS

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

Drop Default Constraint

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

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

3. **UNIQUE CONSTRAINT:** The UNIQUE Constraint prevents two records from having identical values in a column. In the CUSTOMERS table, for example, you might want to prevent two or more people from having an identical age.

Example

For example, the following SQL query creates a new table called CUSTOMERS and adds five columns. Here, the AGE column is set to UNIQUE, so that you cannot have two records with the same age.

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

If the CUSTOMERS 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 CUSTOMERS  
MODIFY AGE INT NOT NULL UNIQUE;
```

You can also use the following syntax, which supports naming the constraint in multiple columns as well.

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

DROP a UNIQUE Constraint

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

```
ALTER TABLE CUSTOMERS  
DROP CONSTRAINT myUniqueConstraint;
```

If you are using MySQL, then you can use the following syntax –

```
ALTER TABLE CUSTOMERS  
DROP INDEX myUniqueConstraint;
```

4. **PRIMARY KEY:** A primary key is a field in a table which 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.

If a table has a primary key defined on any field(s), then you cannot have two records having the same value of that field(s).

**Note** – You would use these concepts while creating database tables.

#### Create Primary Key

Here is the syntax to define the ID attribute as a primary key in a CUSTOMERS table.

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

To create a PRIMARY KEY constraint on the "ID" column when the CUSTOMERS table already exists, use the following SQL syntax –

```
ALTER TABLE CUSTOMER ADD PRIMARY KEY (ID);
```

**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).

For defining a PRIMARY KEY constraint on multiple columns, use the SQL syntax given below.

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

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

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

#### Delete Primary Key

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

```
ALTER TABLE CUSTOMERS DROP PRIMARY KEY ;
```

**5.FOREIGN KEY:** A foreign key is a key used to link two tables together. This is sometimes also called as a referencing key.

A Foreign Key is a column or a combination of columns whose values match a Primary Key in a different table.

**The relationship between 2 tables matches the Primary Key in one of the tables with a Foreign Key in the second table.**

If a table has a primary key defined on any field(s), then you cannot have two records having the same value of that field(s).

### Example

Consider the structure of the following two tables.

#### **CUSTOMERS table**

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

#### **ORDERS table**

```
CREATE TABLE ORDERS (  
  ID INT NOT NULL,  
  DATE DATETIME,  
  CUSTOMER_ID INT references CUSTOMERS(ID),  
  AMOUNT double,  
  PRIMARY KEY (ID)  
);
```

If the ORDERS table has already been created and the foreign key has not yet been set, then use the syntax for specifying a foreign key by altering a table.

```
ALTER TABLE ORDERS  
  ADD FOREIGN KEY (Customer_ID) REFERENCES CUSTOMERS (ID);
```

#### **DROP a FOREIGN KEY Constraint**

To drop a FOREIGN KEY constraint, use the following SQL syntax.

```
ALTER TABLE ORDERS  
  DROP FOREIGN KEY;
```

**6.CHECK CONSTRAINT:** The CHECK Constraint enables a condition to check the value being entered into a record. If the condition evaluates to false, the record violates the constraint and isn't entered into the table.

### Example

For example, the following program creates a new table called CUSTOMERS and adds five columns. Here, we add a CHECK with AGE column, so that you cannot have any CUSTOMER who is below 18 years.

```
CREATE TABLE CUSTOMERS(  
  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)  
);
```

If the CUSTOMERS 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 CUSTOMERS
```



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

You can also use the following syntax, which supports naming the constraint in multiple columns as well –

```
ALTER TABLE CUSTOMERS  
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 CUSTOMERS  
DROP CONSTRAINT myCheckConstraint;
```

## Dropping Constraints

Any constraint that you have defined can be dropped using the ALTER TABLE command with the DROP CONSTRAINT option.

For example, to drop the primary key constraint in the EMPLOYEES table, you can use the following command.

```
ALTER TABLE EMPLOYEES DROP CONSTRAINT EMPLOYEES_PK;
```

Some implementations may provide shortcuts for dropping certain constraints. For example, to drop the primary key constraint for a table in Oracle, you can use the following command.

```
ALTER TABLE EMPLOYEES DROP PRIMARY KEY;
```

Some implementations allow you to disable constraints. Instead of permanently dropping a constraint from the database, you may want to temporarily disable the constraint and then enable it later.

## Integrity Constraints

Integrity constraints are used to ensure accuracy and consistency of the data in a relational database. Data integrity is handled in a relational database through the concept of referential integrity.

There are many types of integrity constraints that play a role in **Referential Integrity (RI)**. These constraints include Primary Key, Foreign Key, Unique Constraints and other constraints which are mentioned above.

- **NULL Values**

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

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

## Syntax

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

```
SQL> CREATE TABLE CUSTOMERS(  
ID INT NOT NULL,  
NAME VARCHAR (20) NOT NULL,
```

```

AGE INT          NOT NULL,
ADDRESS CHAR (25) ,
SALARY DECIMAL (18, 2),
PRIMARY KEY (ID)
);

```

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

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

#### Example

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

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

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	
7	Muffy	24	Indore	

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

```

SQL> SELECT ID, NAME, AGE, ADDRESS, SALARY
FROM CUSTOMERS
WHERE SALARY IS NOT NULL;

```

This would produce the following result –

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

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

```

SQL> SELECT ID, NAME, AGE, ADDRESS, SALARY
FROM CUSTOMERS
WHERE SALARY IS NULL;

```

This would produce the following result –

ID	NAME	AGE	ADDRESS	SALARY
6	Komal	22	MP	

| 7 | Muffy | 24 | Indore | |  
+---+-----+---+-----+---+-----+

- **ALTER TABLE Command**

The SQL **ALTER TABLE** command is used to add, delete or modify columns in an existing table. You should also use the ALTER TABLE command to add and drop various constraints on an existing table.

#### Syntax

The basic syntax of an ALTER TABLE command to add a **New Column** in an existing table is as follows.

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

The basic syntax of an ALTER TABLE command to **DROP COLUMN** in an existing table is as follows.

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

The basic syntax of an ALTER TABLE command to change the **DATA TYPE** of a column in a table is as follows.

```
ALTER TABLE table_name MODIFY COLUMN column_name datatype;
```

The basic syntax of an ALTER TABLE command to add a **NOT NULL** constraint to a column in a table is as follows.

```
ALTER TABLE table_name MODIFY column_name datatype NOT NULL;
```

The basic syntax of ALTER TABLE to **ADD UNIQUE CONSTRAINT** to a table is as follows.

```
ALTER TABLE table_name  
ADD CONSTRAINT MyUniqueConstraint UNIQUE(column1, column2...);
```

The basic syntax of an ALTER TABLE command to **ADD CHECK CONSTRAINT** to a table is as follows.

```
ALTER TABLE table_name  
ADD CONSTRAINT MyUniqueConstraint CHECK (CONDITION);
```

The basic syntax of an ALTER TABLE command to **ADD PRIMARY KEY** constraint to a table is as follows.

```
ALTER TABLE table_name  
ADD CONSTRAINT MyPrimaryKey PRIMARY KEY (column1, column2...);
```

The basic syntax of an ALTER TABLE command to **DROP CONSTRAINT** from a table is as follows.

```
ALTER TABLE table_name  
DROP CONSTRAINT MyUniqueConstraint;
```

If you're using MySQL, the code is as follows –

```
ALTER TABLE table_name  
DROP INDEX MyUniqueConstraint;
```

The basic syntax of an ALTER TABLE command to **DROP PRIMARY KEY** constraint from a table is as follows.

```
ALTER TABLE table_name  
DROP CONSTRAINT MyPrimaryKey;
```

If you're using MySQL, the code is as follows –

```
ALTER TABLE table_name  
DROP PRIMARY KEY;
```

#### Example

Consider the CUSTOMERS table having the following records –

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

Following is the example to ADD a **New Column** to an existing table –

```
ALTER TABLE CUSTOMERS ADD SEX char(1);
```

Now, the CUSTOMERS table is changed and following would be output from the SELECT statement.

ID	NAME	AGE	ADDRESS	SALARY	SEX
1	Ramesh	32	Ahmedabad	2000.00	NULL
2	Ramesh	25	Delhi	1500.00	NULL
3	kaushik	23	Kota	2000.00	NULL
4	kaushik	25	Mumbai	6500.00	NULL
5	Hardik	27	Bhopal	8500.00	NULL
6	Komal	22	MP	4500.00	NULL
7	Muffy	24	Indore	10000.00	NULL

Following is the example to DROP sex column from the existing table.

```
ALTER TABLE CUSTOMERS DROP SEX;
```

Now, the CUSTOMERS table is changed and following would be the output from the SELECT statement.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Ramesh	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	kaushik	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

- **TRUNCATE TABLE**

The SQL **TRUNCATE TABLE** command is used to delete complete data from an existing table.

You can also use DROP TABLE command to delete complete table but it would remove complete table structure from the database and you would need to re-create this table once again if you wish you store some data.

Syntax

The basic syntax of a **TRUNCATE TABLE** command is as follows.

```
TRUNCATE TABLE table_name;
```

#### Example

Consider a CUSTOMERS table having the following records –

```
+---+-----+---+-----+-----+
| 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 |
+---+-----+---+-----+-----+
```

Following is the example of a Truncate command.

```
SQL > TRUNCATE TABLE CUSTOMERS;
```

Now, the CUSTOMERS table is truncated and the output from SELECT statement will be as shown in the code block below –

```
SQL> SELECT * FROM CUSTOMERS;
Empty set (0.00 sec)
```

- **GROUP BY** clause

The SQL **GROUP BY** clause is used in collaboration with the SELECT statement to arrange identical data into groups. This GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.

#### Syntax

The basic syntax of a GROUP BY clause is shown in the following code block. The GROUP BY clause must follow the conditions in the WHERE clause and must precede the ORDER BY clause if one is used.

```
SELECT column1, column2
FROM table_name
WHERE [ conditions ]
GROUP BY column1, column2
ORDER BY column1, column2
```

#### Example

Consider the CUSTOMERS table is having the following records –

```
+---+-----+---+-----+-----+
| 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

If you want to know the total amount of the salary on each customer, then the GROUP BY query would be as follows.

```
SQL> SELECT NAME, SUM(SALARY) FROM CUSTOMERS
      GROUP BY NAME;
```

This would produce the following result –

NAME	SUM(SALARY)
Chaitali	6500.00
Hardik	8500.00
kaushik	2000.00
Khilan	1500.00
Komal	4500.00
Muffy	10000.00
Ramesh	2000.00

Now, let us look at a table where the CUSTOMERS table has the following records with duplicate names –

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Ramesh	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	kaushik	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Now again, if you want to know the total amount of salary on each customer, then the GROUP BY query would be as follows –

```
SQL> SELECT NAME, SUM(SALARY) FROM CUSTOMERS
      GROUP BY NAME;
```

This would produce the following result –

NAME	SUM(SALARY)
Hardik	8500.00
kaushik	8500.00
Komal	4500.00
Muffy	10000.00
Ramesh	3500.00

- **Using VIEWS**

A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query.

A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create a view.

Views, which are a type of virtual tables allow users to do the following –

- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data in such a way that a user can see and (sometimes) modify exactly what they need and no more.
- Summarize data from various tables which can be used to generate reports.

### Creating Views

Database views are created using the **CREATE VIEW** statement. Views can be created from a single table, multiple tables or another view.

To create a view, a user must have the appropriate system privilege according to the specific implementation.

The basic **CREATE VIEW** syntax is as follows –

```
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE [condition];
```

You can include multiple tables in your SELECT statement in a similar way as you use them in a normal SQL SELECT query.

### Example

Consider the CUSTOMERS table having the following records –

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

Following is an example to create a view from the CUSTOMERS table. This view would be used to have customer name and age from the CUSTOMERS table.

```
SQL > CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM CUSTOMERS;
```

Now, you can query CUSTOMERS\_VIEW in a similar way as you query an actual table. Following is an example for the same.

```
SQL > SELECT * FROM CUSTOMERS_VIEW;
```

This would produce the following result.

name	age
Ramesh	32

Khilan	25	
kaushik	23	
Chaitali	25	
Hardik	27	
Komal	22	
Muffy	24	
+-----+-----+		

### The WITH CHECK OPTION

The WITH CHECK OPTION is a CREATE VIEW statement option. The purpose of the WITH CHECK OPTION is to ensure that all UPDATE and INSERTs satisfy the condition(s) in the view definition.

If they do not satisfy the condition(s), the UPDATE or INSERT returns an error.

The following code block has an example of creating same view CUSTOMERS\_VIEW with the WITH CHECK OPTION.

```
CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM CUSTOMERS
WHERE age IS NOT NULL
WITH CHECK OPTION;
```

The WITH CHECK OPTION in this case should deny the entry of any NULL values in the view's AGE column, because the view is defined by data that does not have a NULL value in the AGE column.

### Updating a View

A view can be updated under certain conditions which are given below –

- The SELECT clause may not contain the keyword DISTINCT.
- The SELECT clause may not contain summary functions.
- The SELECT clause may not contain set functions.
- The SELECT clause may not contain set operators.
- The SELECT clause may not contain an ORDER BY clause.
- The FROM clause may not contain multiple tables.
- The WHERE clause may not contain subqueries.
- The query may not contain GROUP BY or HAVING.
- Calculated columns may not be updated.
- All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.

So, if a view satisfies all the above-mentioned rules then you can update that view. The following code block has an example to update the age of Ramesh.

```
SQL > UPDATE CUSTOMERS_VIEW
SET AGE = 35
WHERE name = 'Ramesh';
```

This would ultimately update the base table CUSTOMERS and the same would reflect in the view itself. Now, try to query the base table and the SELECT statement would produce the following result.

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



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

### Inserting Rows into a View

Rows of data can be inserted into a view. The same rules that apply to the UPDATE command also apply to the INSERT command.

Here, we cannot insert rows in the CUSTOMERS\_VIEW because we have not included all the NOT NULL columns in this view, otherwise you can insert rows in a view in a similar way as you insert them in a table.

### Deleting Rows into a View

Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command.

Following is an example to delete a record having AGE = 22.

```
SQL > DELETE FROM CUSTOMERS_VIEW
WHERE age = 22;
```

This would ultimately delete a row from the base table CUSTOMERS and the same would reflect in the view itself. Now, try to query the base table and the SELECT statement would produce the following result.

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
7	Muffy	24	Indore	10000.00

### Dropping Views

Obviously, where you have a view, you need a way to drop the view if it is no longer needed. The syntax is very simple and is given below –

```
DROP VIEW view_name;
```

Following is an example to drop the CUSTOMERS\_VIEW from the CUSTOMERS table.

```
DROP VIEW CUSTOMERS_VIEW;
```

### Queries Using aggregate functions :-

#### 1. 2<sup>nd</sup> highest Salary :

**Syntax:**

```
SELECT MAX(SALARY)
```

```
FROM TABLENAME
```

```
WHERE SALARY NOT IN ( SELECT MAX(SALARY) FROM TABLENAME);
```

#### 2. 2<sup>ND</sup> minimum Salary

**Syntax:**

```
SELECT MIN(SALARY)
```

```
FROM TABLENAME
```

```
WHERE SALARY NOT IN ( SELECT MIN(SALARY) FROM TABLENAME);
```

#### 3. Nth Highest Salary:

**Syntax:**

```
SELECT MIN(SALARY)
```

```
FROM TABLENAME
```

```
WHERE SALARY IN (SELECT TOP Nth SALARY FROM TABLENAME  
ORDER BY SALARY DESC);
```

#### 4. Nth Minimum Salary :

**Syntax:**

```
SELECT MAX (SALARY)
```

```
FROM TABLENAME
```

```
WHERE SALARY IN (SELECT TOP Nth SALARY FROM TABLENAME  
ORDER BY SALARY);
```

## IN Operator

The **IN** operator allows you to specify multiple values in a **WHERE** clause.

The **IN** operator is a shorthand for multiple **OR** conditions.

IN Syntax

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (value1, value2, ...);
```

Or

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (SELECT STATEMENT);
```

For ex: The following SQL statement selects all customers that are located in "Germany", "France" or "UK":

```
SELECT * FROM Customers
WHERE Country IN ('Germany', 'France', 'UK');
```

Ex 2: The following SQL statement selects all customers that are NOT located in "Germany", "France" or "UK":

```
SELECT * FROM Customers
WHERE Country NOT IN ('Germany', 'France', 'UK');
```

## The SQL BETWEEN Operator

The **BETWEEN** operator selects values within a given range. The values can be numbers, text, or dates.

The **BETWEEN** operator is inclusive: begin and end values are included.

BETWEEN Syntax

```
SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

Ex: The following SQL statement selects all products with a price between 10 and 20:

```
SELECT * FROM Products
WHERE Price BETWEEN 10 AND 20;
```

### BETWEEN Text Values Example

The following SQL statement selects all products with a ProductName between Carnarvon Tigers and Mozzarella di Giovanni:

Ex:

```
SELECT * FROM Products
WHERE ProductName BETWEEN 'Carnarvon Tigers' AND 'Mozzarella di Giovanni'
ORDER BY ProductName;
```

## Union and Union ALL

### UNION

The UNION command is used to select related information from two tables, which is like a JOIN command. However, when using UNION command, all the selected columns need to be of the same data type. With UNION, only distinct values are selected.

### UNION ALL

UNION ALL command is equal to UNION command, except that UNION ALL selects all the values.

The difference between Union and Union all is that Union all will not eliminate duplicate rows, instead it just pulls all the rows from all the tables fitting your query specifics and combines them into a table.

*A UNION statement effectively does a SELECT DISTINCT on the results set. If you know that all the records returned are unique from your union, use UNION ALL instead, it gives faster results.*

### **Example**

Table 1 : First,Second,Third,Fourth,Fifth

Table 2 : First,Second,Fifth,Sixth

### **Result Set**

UNION: First,Second,Third,Fourth,Fifth,Sixth (This will remove duplicate values)

UNION ALL: First,First,Second,Second,Third,Fourth,Fifth,Fifth,Sixth,Sixth (This will repeat values)

## The SQL COUNT(), AVG() and SUM() Functions

The **COUNT()** function returns the number of rows that matches a specified criterion.

COUNT() Syntax

```
SELECT COUNT(column_name)  
FROM table_name  
WHERE condition;
```

COUNT() Example

The following SQL statement finds the number of products:

```
SELECT COUNT(ProductID)  
FROM Products;
```

The **AVG()** function returns the average value of a numeric column.

AVG() Syntax

```
SELECT AVG(column_name)  
FROM table_name  
WHERE condition;
```

AVG() Example

The following SQL statement finds the average price of all products:

```
SELECT AVG(Price)  
FROM Products;
```

The **SUM()** function returns the total sum of a numeric column.

SUM() Syntax

```
SELECT SUM(column_name)  
FROM table_name  
WHERE condition;
```

SUM() Example

The following SQL statement finds the sum of the "Quantity" fields in the "OrderDetails" table:

```
SELECT SUM(Quantity)
FROM OrderDetails;
```

## The SQL MIN() and MAX() Functions

The **MIN()** function returns the smallest value of the selected column.

The **MAX()** function returns the largest value of the selected column.

### MIN() Syntax

```
SELECT MIN(column_name)
FROM table_name
WHERE condition;
```

#### **Min Example**

The following SQL statement finds the price of the cheapest product:

```
SELECT MIN(Price) AS SmallestPrice
FROM Products;
```

### MAX() Syntax

```
SELECT MAX(column_name)
FROM table_name
WHERE condition;
```

### MAX() Example

The following SQL statement finds the price of the most expensive product:

```
SELECT MAX(Price) AS LargestPrice
FROM Products;
```

## The FIRST() Function

The FIRST() function returns the first value of the selected column.

### SQL FIRST() Syntax

```
SELECT FIRST(column_name) FROM table_name;
```

## SQL FIRST() Example

The following SQL statement selects the first value of the "CustomerName" column from the "Customers" table:

```
SELECT FIRST(CustomerName) AS FirstCustomer FROM Customers; LAST
```

## The LAST() Function

The LAST() function returns the last value of the selected column.

### SQL LAST() Syntax

```
SELECT LAST(column_name) FROM table_name;
```

### SQL LAST() Example

The following SQL statement selects the last value of the "CustomerName" column from the "Customers" table:

```
SELECT LAST(CustomerName) AS LastCustomer FROM Customers;
```

## The HAVING Clause

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

Having:- used to select records which satisfy the given condition and also it is used with aggregate function.

### SQL HAVING Syntax

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

## SQL HAVING Examples

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

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

## SQL Views

### SQL CREATE VIEW Statement

In SQL, a view is a virtual table based on the result-set of an SQL statement.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can add SQL statements and functions to a view and present the data as if the data were coming from one single table.

A view is created with the **CREATE VIEW** statement.

VIEW: CREATES A VIRTUAL TABLE BASED ON RESULT SET OF SQL STATEMENT.

### CREATE VIEW Syntax

```
CREATE VIEW view_name AS  
SELECT column1, column2, ...  
FROM table_name  
WHERE condition;
```

### SQL CREATE VIEW Examples

The following SQL creates a view that shows all customers from Brazil:

```
CREATE VIEW [Brazil Customers] AS  
SELECT CustomerName, ContactName  
FROM Customers  
WHERE Country = 'Brazil';
```

We can query the view above as follows:

```
SELECT * FROM [Brazil Customers];
```

TABLE 1    TID, TNAME, TCITY,    TID - PK

TABLE 2    TDATE, T2ID2, T2NAME, TID    TID-FK, T2ID2 - PK

**CREATE BY - ATUL KUMAR (LINKEDIN)**