Contents

[SELECT: 4](#_Toc187430134)

[FROM: 4](#_Toc187430135)

[TOP (5): 4](#_Toc187430136)

[PERCENT: 4](#_Toc187430137)

[SELECT \*: 4](#_Toc187430138)

[AS: 5](#_Toc187430139)

[DISTINCT: 5](#_Toc187430140)

[Order By: 5](#_Toc187430141)

[Defining criteria 5](#_Toc187430142)

[<>: 6](#_Toc187430143)

[Where total BETWEEN 10 AND 30: 6](#_Toc187430144)

[Selecting and excluding NULL values: 6](#_Toc187430145)

[And if we want to exclude null data then: 6](#_Toc187430146)

[Several Condtions: 7](#_Toc187430147)

[We can also use OR between them. 7](#_Toc187430148)

[IN: 7](#_Toc187430149)

[LIKE: 8](#_Toc187430150)

[WHERE name LIKE 'A\_n'; 8](#_Toc187430151)

[SELECT \* FROM employees 8](#_Toc187430152)

[Aggregations in SQL 9](#_Toc187430153)

[SUM: 9](#_Toc187430154)

[COUNT: 9](#_Toc187430155)

[MIN or MAX: 10](#_Toc187430156)

[AVG Function in SQL 10](#_Toc187430157)

[LEN: 10](#_Toc187430158)

[LEFT and RIGHT Functions in SQL 10](#_Toc187430159)

[CHARINDEX: 10](#_Toc187430160)

[SUBSTRING Function 11](#_Toc187430161)

[REPLACE Function 11](#_Toc187430162)

[GROUP BY 11](#_Toc187430163)

[HAVING Clause in SQL 11](#_Toc187430164)

[Primary Key 13](#_Toc187430165)

[Foreign Key 13](#_Toc187430166)

[Getting data from two different tables: 14](#_Toc187430167)

[What is a ? 14](#_Toc187430168)

[Comparison Between Inner Join and Left Join 15](#_Toc187430169)

[UNION: 16](#_Toc187430170)

[Union All: 16](#_Toc187430171)

[Create: 17](#_Toc187430172)

[Data Types that can be stored in columns: 17](#_Toc187430173)

[INSERT INTO 17](#_Toc187430174)

[INSERT INTO ... SELECT 18](#_Toc187430175)

[Update: 18](#_Toc187430176)

[Delete: 18](#_Toc187430177)

[Truncate: 19](#_Toc187430178)

[DECLARE: 19](#_Toc187430179)

[SET: 19](#_Toc187430180)

[Schema in Database: 19](#_Toc187430181)

[Avg, Mix, Max: 19](#_Toc187430182)

[Blank Value vs Null Value: 20](#_Toc187430183)

[Replacing Null value with some Value: 20](#_Toc187430184)

[Handle Null values with coalesce function: 21](#_Toc187430185)

[Changing column values with CASE 21](#_Toc187430186)

[Count and distinct count: 22](#_Toc187430187)

[DateAdd: 22](#_Toc187430188)

[DATEDIFF 23](#_Toc187430189)

[Round: 23](#_Toc187430190)

[Using Truncate for Decimals: 24](#_Toc187430191)

[Absolute, Square, Square Root, Log: 24](#_Toc187430192)

[Declaring Variable: 25](#_Toc187430193)

[While Loops: 25](#_Toc187430194)

[Syntax of While Loop: 26](#_Toc187430195)

[Using break statement in While Loop: 26](#_Toc187430196)

[Derived Table: 26](#_Toc187430197)

[Common Table Expression 27](#_Toc187430198)

[Window Function: 27](#_Toc187430199)

[Common Window Function: 28](#_Toc187430200)

[Inner Join: 28](#_Toc187430201)

[One-to-many relationships 28](#_Toc187430202)

[One to one relationship: 28](#_Toc187430203)

[Many to Many relationship: 28](#_Toc187430204)

[Double Inner Join: 29](#_Toc187430205)

[Left Join: 29](#_Toc187430206)

[Full Join: 29](#_Toc187430207)

[Cross Join: 30](#_Toc187430208)

[Self-Join: 30](#_Toc187430209)

[Union and Union All: 30](#_Toc187430210)

[Intersect: 31](#_Toc187430211)

[Except: 32](#_Toc187430212)

[Semi Join: 32](#_Toc187430213)

SQL stands for Structured Query Language.

# SELECT:

It is used to retrieve data from database.

SELECT name

FROM students;

# FROM:

It is used to mention the table from where we want to get the columns will all rows.

SELECT age

FROM students;

# TOP (5):

Top is used to limit the first rows till how much we want to retrieve data from rows. The parameter 5 means that we need top 5 rows from the column mentioned.

SELECT TOP (5) name

FROM students;

# PERCENT:

If we mention the percent keyword after top so instead of 5 rows, it’ll return 5% rows from top.

SELECT TOP (10) PERCENT name

FROM students;

# SELECT \*:

Using \* with select means that return all columns and rows from the required table.

SELECT \*

FROM students;

# AS:

It’s used for aliasing. Used to give the selected column a temporary alias (alternative name) for the output. The alias doesn’t change the column name in the database it’s just for readability in the result.

SELECT name AS student\_name

FROM students;

# DISTINCT:

The DISTINCT keyword is applied after SELECT. It filters out duplicate rows based on the columns listed in the SELECT statement.

SELECT DISTINCT age

FROM students;

# Order By:

Queries return set of data. And its not necessary that the data which we’ll get will be in the same sequence always. That’s why we use ORDER keyword to get the data in a specific order always.

The given below code will order in ascending. You want in descending then you can add DESC keyword after column name

SELECT TOP(10) PERCENT name

FROM students

Order By name, id

Specifying multiple columns in an ORDER BY clause allows you to define a more detailed sorting order. The database will sort the results **first** by the first column, and then **within rows that have the same value for the first column**, it will sort by the second column.

SELECT TOP(10) PERCENT name

FROM students

Order By name DESC, id

This ordering can be applied to columns having text as well.

# Defining criteria

We can define criteria with some conditions as well to get data from database.

Select name

From students

Where age > 15;

Or Where date = ’12-2-2023’

# <>:

In sql <> means those values those are either bigger or smaller than mentioned number and not equal to.

# Where total BETWEEN 10 AND 30:

In this query between and “and” are used collectively to select data between a specific range.

If we want to exclude that range from our required data than we can use

Where total NOT BETWEEN 10 AND 30

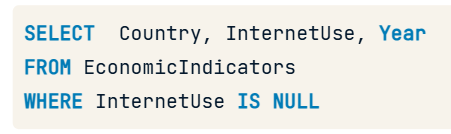
# Selecting and excluding NULL values:

For selecting those values from column where the value is null we can use this code

Select name

From students

Where name IS NULL;

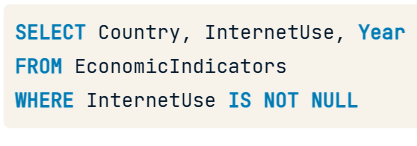


# And if we want to exclude null data then:

Select name

From students

Where name IS NOT NULL;



# Several Condtions:

We can apply several conditions using AND between condtions.

Select name

From students

Where age > 15

AND gender = ‘Male’;

# We can also use OR between them.

Select name

From students

Where age > 15

OR gender = ‘Male’;

# IN:

The IN keyword in SQL is used to filter records where a column's value matches any value in a specified list. It acts as a shorthand for multiple OR conditions.

SELECT \* FROM employees

WHERE department IN ('HR', 'Finance', 'IT');

Returns all employees who work in the HR, Finance, or IT departments.

# LIKE:

The LIKE keyword in SQL is used in a WHERE clause to search for a specified pattern in a column. It is commonly used with wildcard characters:

**Wildcards**:

* %: Represents zero, one, or multiple characters.
* \_: Represents a single character.

SELECT \* FROM employees

WHERE name LIKE 'J%';

Returns all employees whose names start with "J".

SELECT \* FROM employees

# WHERE name LIKE 'A\_n';

Returns all employees whose names have "A" as the first letter, followed by any one character, and end with "n" (e.g., "Ann").

# SELECT \* FROM employees

WHERE name LIKE '%son%';

Returns all employees whose names contain "son" anywhere.

# Aggregations in SQL

# SUM:

The SUM function in SQL calculates the total (sum) of a numeric column. It is often used to perform aggregations in combination with GROUP BY or on its own.

SELECT SUM(salary) AS total\_salary

FROM employees;

This query calculates the total salary of all employees.

But if we want to sum several columns so its necessary to add several sum keywords otherwise we’ll get an error. And it necessary to add AS and a temporary name because when the sum is returned there will be no column name with it.

# COUNT:

The COUNT function in SQL is used to count the number of rows in a table or the number of non-NULL values in a column.

SELECT COUNT(\*) AS total\_rows

FROM employees;

This query counts the total number of rows in the employees table, including rows with NULL values.

SELECT COUNT(salary) AS total\_salaries

FROM employees;

This query counts the number of non-NULL values in the salary column.

If we want to get unique values in result then:

SELECT COUNT(DISTINCT department) AS unique\_departments

FROM employees;

This query counts the number of unique departments in the department column.

# MIN or MAX:

The MIN function in SQL is used to return the smallest (minimum) value in a column. It is commonly used to find the lowest value in a dataset, such as the lowest salary or earliest date.

SELECT MIN(salary) AS minimum\_salary

FROM employees;

# AVG Function in SQL

The AVG function in SQL calculates the average (mean) of a numeric column. It is useful for finding average values, such as average salaries, prices, or scores in a dataset.

SELECT AVG(salary) AS average\_salary

FROM employees;

This query calculates the average salary of all employees.

# LEN:

The LEN function in SQL is used to calculate the **length** of a string (the number of characters in a given text). It is often used to analyze text data, such as finding the length of names, codes, or other string-based columns.

SELECT LEN(name) AS name\_length

FROM employees;

# LEFT and RIGHT Functions in SQL

The LEFT and RIGHT functions are used to extract a specified number of characters from the beginning or the end of a string, respectively. These functions are helpful in text manipulation, such as extracting prefixes, suffixes, or parts of a string.

SELECT LEFT(name, 3) AS first\_three\_characters

FROM employees;

# CHARINDEX:

The CHARINDEX function in SQL is used to find the starting position of a substring within a string. It is case-insensitive in most SQL implementations and is commonly used to locate specific characters or words in a text column.

SELECT CHARINDEX('SQL', 'Learn SQL with ease') AS position;

# SUBSTRING Function

The SUBSTRING function extracts a portion of a string, starting from a specified position and for a specified length.

SELECT SUBSTRING('Hello World', 7, 5) AS extracted\_string;

(Extracts 5 characters starting from the 7th position.)

# REPLACE Function

The REPLACE function is used to replace all occurrences of a specified substring within a string with another substring.

REPLACE(column\_name, 'old\_string', 'new\_string')

# GROUP BY

The **GROUP BY** clause in SQL is used to group rows that have the same values in specified columns into aggregate data. It is often used with aggregate functions like SUM, COUNT, AVG, MIN, and MAX to perform operations on each group of rows.

SELECT department, COUNT(\*) AS employee\_count

FROM employees

GROUP BY department;

# HAVING Clause in SQL

The **HAVING** clause in SQL is used to filter groups of data that are created using the **GROUP BY** clause. It allows you to apply conditions to aggregated data (e.g., results of SUM, COUNT, AVG, etc.).

**Key Points**

1. **Used with GROUP BY**:
   * HAVING is applied to groups of rows after the aggregation.
   * Similar to the WHERE clause, but it works on aggregate data.
2. **Difference Between WHERE and HAVING**:
   * **WHERE**: Filters rows **before** grouping and aggregation.
   * **HAVING**: Filters groups **after** aggregation.

SELECT region, SUM(sales\_amount) AS total\_sales

FROM sales

GROUP BY region

HAVING SUM(sales\_amount) > 3000;

# Primary Key

1. **Definition**: A column (or a set of columns) in a table that uniquely identifies each row in that table.
2. **Characteristics**:
   * **Unique**: The values in the primary key column(s) must be unique for each row.
   * **Non-Nullable**: It cannot contain NULL values.
   * **Single Table**: Each table can have only one primary key.
3. **Example**:

sql

Copy code

CREATE TABLE Students (

StudentID INT PRIMARY KEY, -- Primary Key

Name VARCHAR(50),

Age INT

);

Here, StudentID uniquely identifies each student.

# Foreign Key

1. **Definition**: A column (or a set of columns) in one table that refers to the **primary key** in another table, establishing a link between the two tables.
2. **Characteristics**:
   * Ensures **referential integrity** by enforcing that the value in the foreign key column matches a value in the referenced primary key column.
   * A table can have multiple foreign keys.
3. **Example**:

sql

Copy code

CREATE TABLE Enrollments (

EnrollmentID INT PRIMARY KEY,

StudentID INT,

CourseID INT,

FOREIGN KEY (StudentID) REFERENCES Students(StudentID)

);

* + Here, StudentID in Enrollments references StudentID in Students.
  + This ensures that every StudentID in Enrollments exists in the Students table.

# Getting data from two different tables:

SELECT

album\_id, -- The unique identifier for each album.

title, -- The title of the album.

album.artist\_id,-- The artist ID from the album table.

name AS artist\_name -- The name of the artist, renamed as "artist\_name" for clarity.

FROM album

INNER JOIN artist

SELECT *column\_name(s)*  
FROM *table1*  
INNER JOIN *table2*ON *table1.column\_name*=*table2.column\_name*;

The SELECT clause specifies the columns to be retrieved from the tables:

* album\_id, title, and artist\_id come from the album table.
* name AS artist\_name comes from the artist table, where name is renamed to artist\_name for readability in the output.
* **FROM album**: Specifies the main table (album) to query.
* **INNER JOIN artist**: Combines the album table with the artist table based on a matching condition:
* This means rows from both tables are matched where the artist\_id in the album table is equal to the artist\_id in the artist table.
* Only rows that satisfy this condition are included in the result (because of the INNER JOIN).

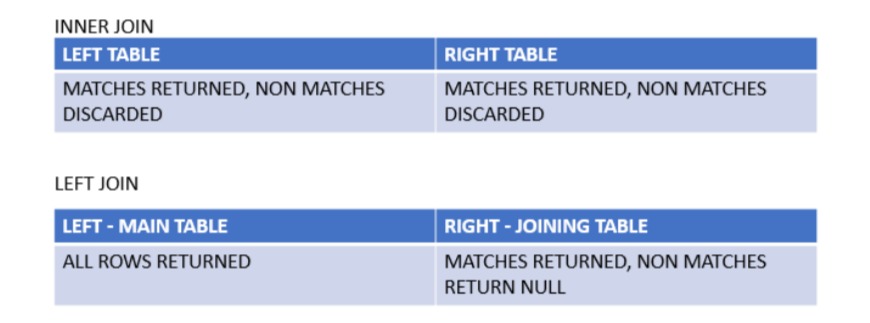
# What is a ?

SELECT *column\_name(s)*  
FROM *table1*  
LEFT JOIN *table2*ON *table1.column\_name*=*table2.column\_name*;

* A **LEFT JOIN** returns **all rows from the left table** (in this case, the Admitted table), and the **matching rows from the right table** (Discharged table).
* If there is no match in the right table, the result will include NULL for the columns from the right table

# Comparison Between Inner Join and Left Join

* **Row Inclusion**:
  + **INNER JOIN** includes only matching rows from both tables.
  + **LEFT JOIN** includes all rows from the left table, with NULL for non-matching rows in the right table.
* **Non-Matching Rows**:
  + **INNER JOIN** excludes non-matching rows from both tables.
  + **LEFT JOIN** keeps non-matching rows from the left table.
* **Usage**:
  + **INNER JOIN** is used when you only need matching data between tables.
  + **LEFT JOIN** is used when you need all data from the left table, regardless of matches.
* **Result Size**:
  + **INNER JOIN** typically produces fewer rows.
  + **LEFT JOIN** may produce more rows if unmatched rows exist in the left table.



# UNION:

The **UNION** operator in SQL is used to combine the results of two or more SELECT queries into a single result set. It removes duplicate rows by default.

SELECT column1, column2

FROM table1

UNION

SELECT column1, column2

FROM table2;

# Union All:

The **UNION ALL** operator in SQL is similar to **UNION**, but it **does not remove duplicate rows**. It combines the results of two or more SELECT queries and includes **all rows**, even if some are duplicates.

CRUD Operations:

CRUD is combination of 4 words, create, read, update and delete.

# Create:

Create Table keyword is used to create a table. The table should have a unique name. for making columns, we have to define 3 things in parenthesis.

* 1. Column name
  2. Column data type
  3. Size (where necessary)

Create table test\_table (

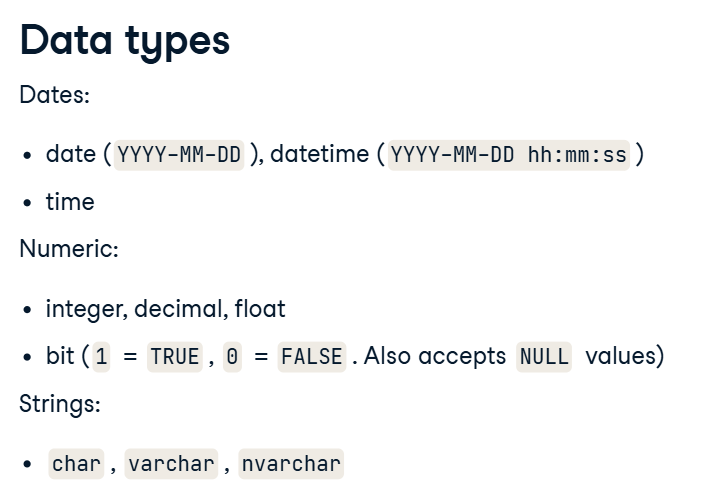
Test\_date date,

Test name varchar (20),

Test\_num int

)

# Data Types that can be stored in columns:



# INSERT INTO

The INSERT INTO statement is used to add new rows (records) into a table in a database. This is helpful when you want to add specific values manually or programmatically to a table.

**Syntax:**

INSERT INTO table\_name (column1, column2, column3, ...)

VALUES (value1, value2, value3, ...);

# INSERT INTO ... SELECT

The INSERT INTO ... SELECT statement is used to copy data from one table to another. It allows you to select data from a source table and insert it into a target table, which is useful when migrating or transforming data.

**Syntax:**

INSERT INTO target\_table (column1, column2, column3, ...)

SELECT column1, column2, column3, ...

FROM source\_table

WHERE condition;

# Update:

The **UPDATE** statement in SQL is used to modify existing records in a table. This is helpful when you want to change specific values or multiple values in one or more rows based on certain conditions.

**Syntax**

UPDATE table\_name

SET column1 = value1, column2 = value2, ...

WHERE condition;

# Delete:

The **DELETE** statement in SQL is used to remove one or more rows from a table. You can specify a condition to delete specific rows or omit the condition to delete all rows in the table (use this with caution!).

**Syntax**

DELETE FROM table\_name

WHERE condition;

# Truncate:

The **TRUNCATE** statement in SQL is used to **remove all rows from a table** quickly and efficiently. Unlike the DELETE statement, TRUNCATE does not allow filtering with a WHERE clause—it always removes all rows.

**Syntax**

TRUNCATE TABLE table\_name;

# DECLARE:

The DECLARE statement is used to **define variables** in SQL. These variables can hold temporary data, which can be used and manipulated during the execution of a script or procedure.

**Syntax**

DECLARE @variable\_name datatype;

# SET:

The SET statement is used to **assign a value** to a variable declared with DECLARE.

**Syntax**

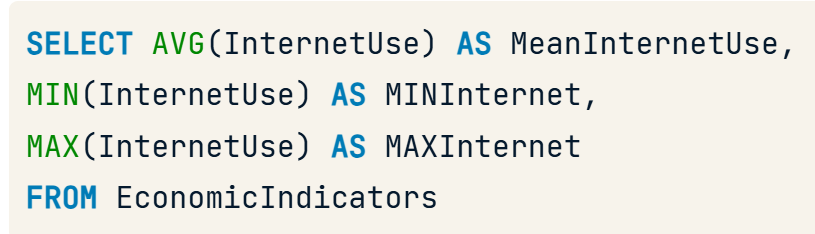
SET @variable\_name = value;

# Schema in Database:

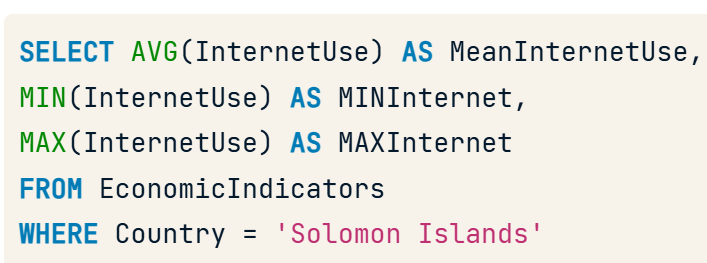
Schema is Logical Representation of tables and data.

# Avg, Mix, Max:

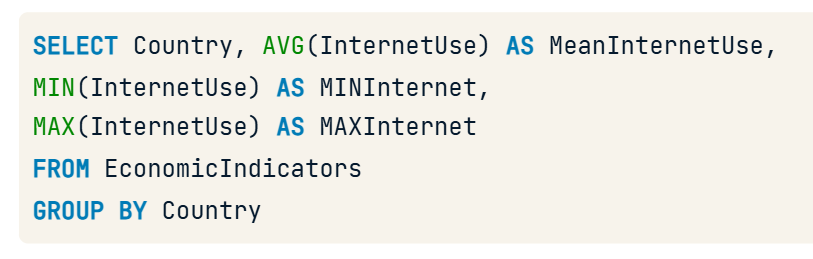
Finding min, max and avg generally for columns.



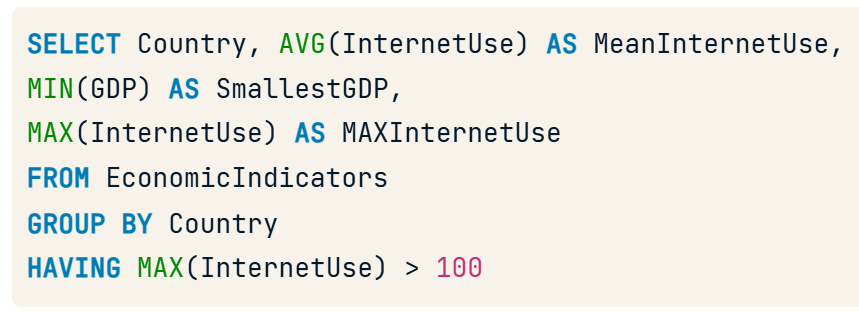
And if we want to add any specific condition we can use where keyword:



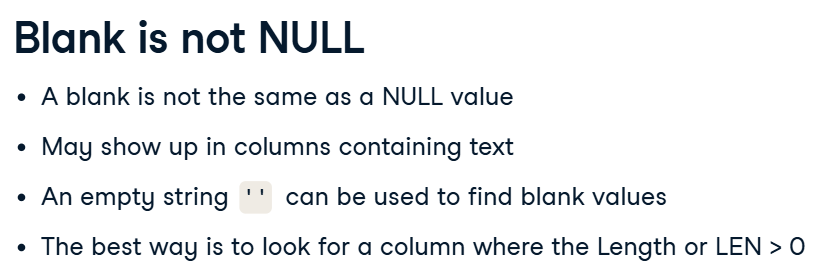
And if we want to find these 3 values from different groups we can add groupby keyword:



But remember that we cannot use where with the groupby, instead we can use having in the following way:



# Blank Value vs Null Value:

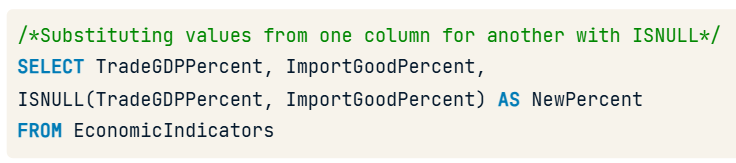


# Replacing Null value with some Value:

We can replace a null value with a specific value everywhere with the following code:



If we want to borrow value for null values from a column so we can use the following code:



# Handle Null values with coalesce function:

The COALESCE function in T-SQL is used to return the first non-NULL value from a list of expressions. It checks each expression in order and stops at the first one that is not NULL. If all expressions are NULL, it returns NULL. This function is useful for handling missing data or providing default values.

For example,

SELECT COALESCE(Address, 'No Address Provided') AS AddressInfo

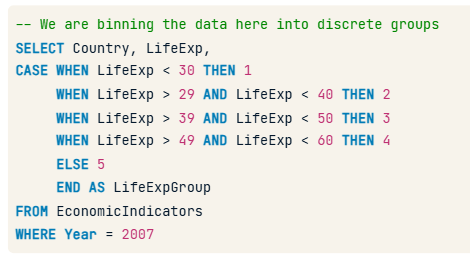
FROM Customers;

This will return the value in the Address column if it is not NULL; otherwise, it will display 'No Address Provided'.



# Changing column values with CASE

The CASE statement allows us to evaluate records like an if statement. We can use the CASE statement to check if a column contains a value and WHEN it does THEN we can replace the value with some other value of our choice, ELSE replace it with any other default value. A CASE statement must have at least four keywords: CASE, WHEN, THEN, and END. The ELSE is optional, but it often makes sense to include it.



# Count and distinct count:

If you want to count all values from a table then use count:

Select count(\*) from table;

But if you want to just know distinct values of a column than use this code:

Select count(distinct column\_name)

# DateAdd:

The DATEADD function in T-SQL is used to add or subtract a specified amount of time to a date or datetime value. It allows you to manipulate date values by adding or subtracting intervals like days, months, years, hours, or minutes. The function takes three arguments: the date part (e.g., YEAR, MONTH, DAY), the number of units to add (positive to add, negative to subtract), and the date or datetime value to adjust. This is useful for calculating future or past dates, scheduling events, or adjusting timestamps in queries.

For example, to add 10 days to a date, you can use:

SELECT DATEADD(dd, 10, '2024-01-01') AS NewDate;

, which returns '2024-01-11'. To subtract 2 months, you can use:

SELECT DATEADD(mm, -2, '2024-03-01') AS NewDate;

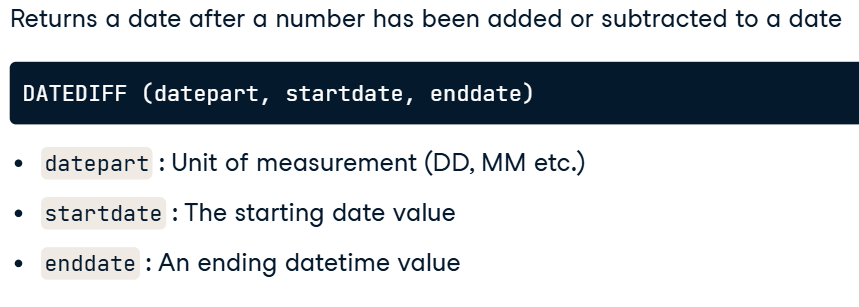
which returns '2024-01-01'. You can also combine it with other functions, like GETDATE(), to work with the current date. For instance, SELECT

DATEADD(yy, 1, GETDATE()) AS NextYear;

adds 1 year to the current date.

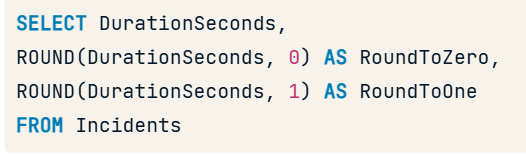
# DATEDIFF

Similar to DATEADD, DATEDIFF takes three required arguments. The first argument is the same, datepart, a unit of measurement. The second and third arguments are the start and end dates, respectively.

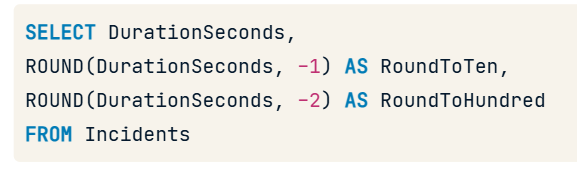


# Round:

The ROUND function lets you round numbers on either side of the decimal. It takes two required arguments, the number to be rounded, and length, the number of places the number should be rounded. If the length specified is negative, the numbers on the left side of the decimal, that is the whole numbers are rounded. If the number is positive, the numbers on the right side of the decimal, the decimal numbers are rounded.

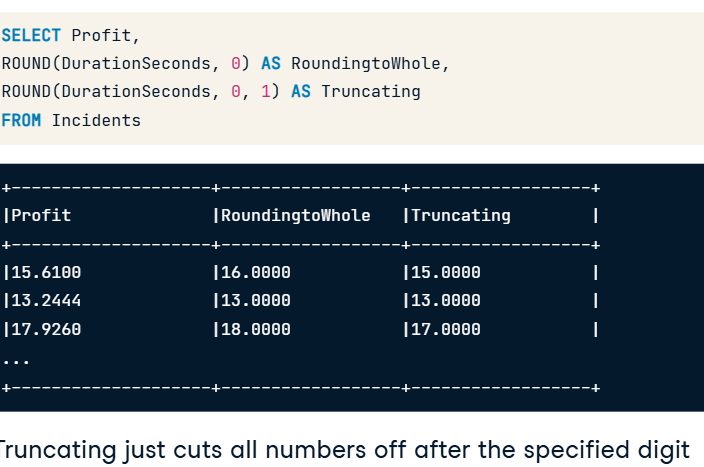


If you want to round to tens, hundred so you can use the following code:



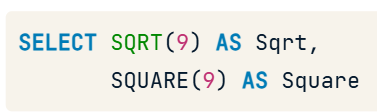
# Using Truncate for Decimals:

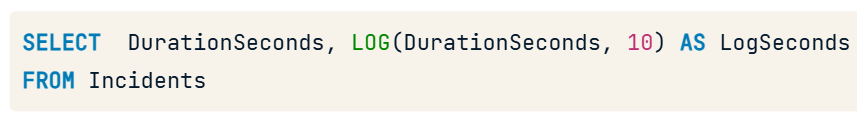
In SQL, the TRUNCATE function is used to truncate a numeric value to a specified number of decimal places without rounding. It removes the digits after the specified precision, effectively shortening the number to the desired length. For example, TRUNCATE(123.4567, 2) will return 123.45, as it cuts off the number after two decimal places without altering the last digit. It is particularly useful in cases where precise control over the number of decimals is required, without introducing rounding errors that might occur with functions like ROUND.



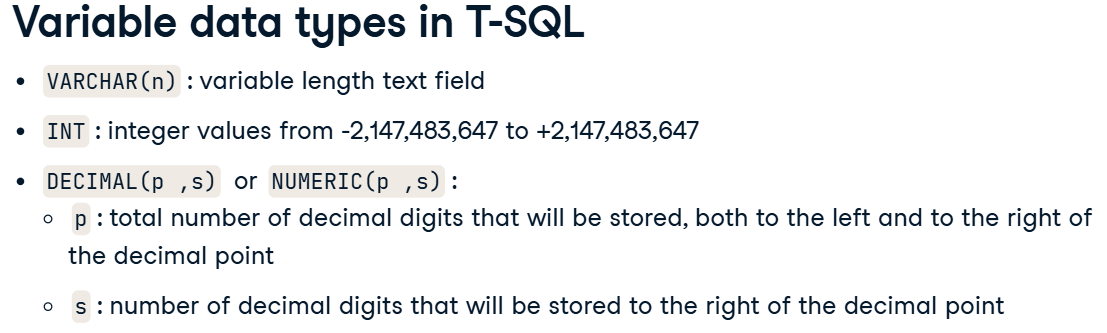
# Absolute, Square, Square Root, Log:

There are times that you do not care if the value is positive or negative, you are only interested in the value of the number. When that is the case, you can use ABS to return only non-negative values. 

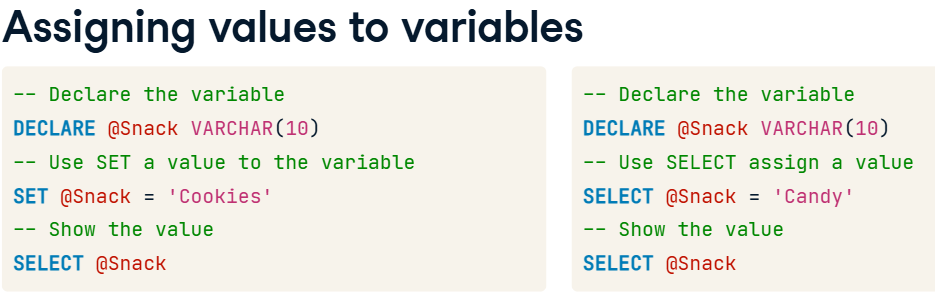




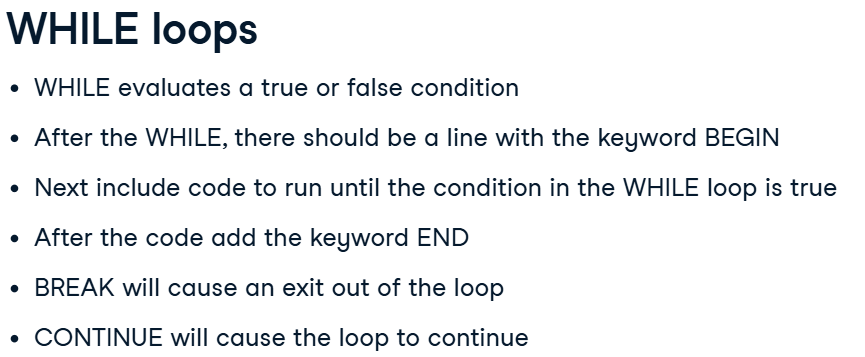
# Declaring Variable:



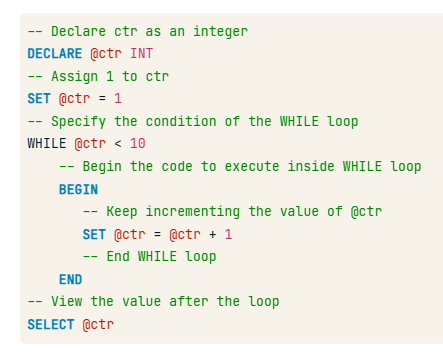
For assigning values to variable, one of the following ways can be used:



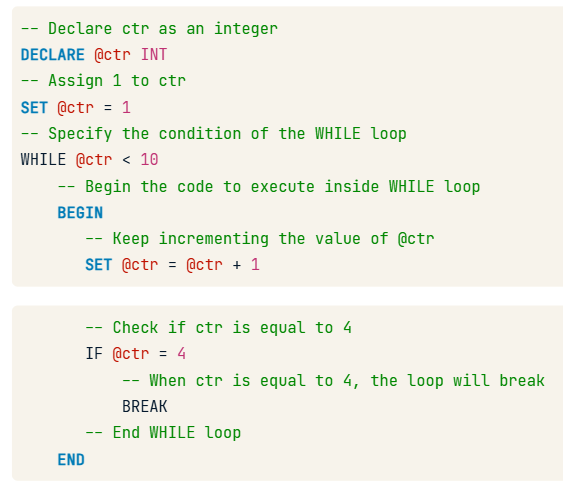
# While Loops:



## Syntax of While Loop:



## Using break statement in While Loop:



# Derived Table:

A **derived table** in SQL is a temporary table created within a query by using a subquery in the FROM clause. It allows you to use the result of a subquery as a virtual table, enabling further processing in the outer query. Derived tables are useful for simplifying complex queries by breaking them into smaller, more manageable components.

Example:

SELECT d.department\_name, AVG(d.salary) AS average\_salary

FROM (

SELECT department\_id, salary

FROM employees

) d

GROUP BY d.department\_name;

In this query, the subquery inside the FROM clause creates a derived table d, which is then used in the outer query to calculate the average salary by department. Derived tables exist only during the execution of the query and do not persist in the database.

# Common Table Expression

A **CTE (Common Table Expression)** in SQL is a temporary result set that you can define and reference within a single SQL query. It is created using the WITH keyword and is especially useful for simplifying complex queries, improving readability, and reusing the same result set multiple times in a query.

WITH EmployeeCTE AS (

SELECT department\_id, AVG(salary) AS average\_salary

FROM employees

GROUP BY department\_id

)

SELECT \*

FROM EmployeeCTE

WHERE average\_salary > 50000;

# Window Function:

A **window function** in SQL is a function that performs a calculation across a set of rows related to the current row, defined by a "window" or partition. Unlike aggregate functions (e.g., SUM, AVG), which collapse rows into a single result, window functions retain individual rows while adding calculated values.

SELECT

employee\_id,

department\_id,

salary,

RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS rank\_in\_dept

FROM employees;

## Common Window Function:

**Ranking Functions**: ROW\_NUMBER(), RANK(), DENSE\_RANK(), NTILE()

**Aggregate Functions**: SUM(), AVG(), MAX(), MIN(), COUNT()

**Analytic Functions**: LAG(), LEAD(), FIRST\_VALUE(), LAST\_VALUE()

# Inner Join:

An **INNER JOIN** in SQL is used to combine rows from two or more tables based on a related column between them. It retrieves only the rows that have matching values in both tables, ensuring the output includes only the common data. This is particularly useful for finding relationships between datasets. For instance, if you have a Customers table and an Orders table, an INNER JOIN can show which customers have placed orders.

SELECT Customers.CustomerID, Customers.Name, Orders.OrderID

FROM Customers

INNER JOIN Orders ON Customers.CustomerID = Orders.CustomerID;

In this query, the INNER JOIN matches rows in the Customers table with rows in the Orders table where the CustomerID is the same, returning a list of customers who have placed orders.

# One-to-many relationships

The first type of relationship we'll talk about is a one-to-many relationship. This is the most common type of relationship, one where a single entity can be associated with several entities. Think about a music library. One artist can produce many songs over their career. This is a one-to-many relationship. The same applies for authors and their books, directors and movie titles, and so on.

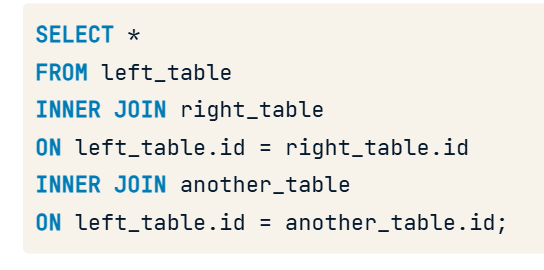
# One to one relationship:

A second type of relationship is a one-to-one relationship. One-to-one relationships imply unique pairings between entities and are therefore less common. A commonly held premise of forensic science is that no two fingerprints are identical, and therefore that a particular fingerprint can only be generated by one person. This is an example of a one-to-one relationship: one fingerprint for one finger.

# Many to Many relationship:

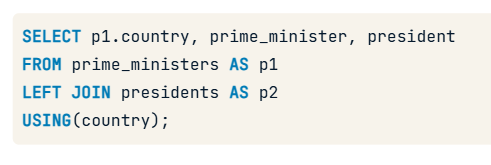
The last type of relationship we'll discuss is a many-to-many relationship. An example of this is languages and countries. Here we show the official languages of Germany, Belgium and the Netherlands, where we see that many languages can be spoken in many countries. For example, Belgium has three official languages: French, German, and Dutch. Conversely, languages can be official in many countries: Dutch is an official language of both the Netherlands and Belgium, but not Germany.

# Double Inner Join:



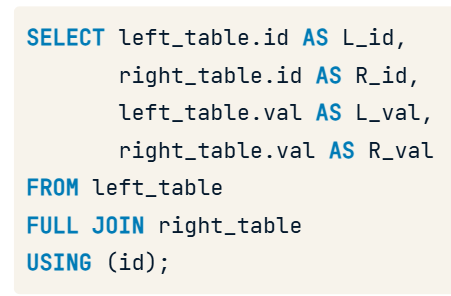
# Left Join:

LEFT JOIN will return all records in the left\_table, and those records in the right\_table that match on the joining field provided. In the diagram shown, the values of 2 and 3 do not appear in the id field of right\_table but will still be retained in the join.



# Full Join:

A **full join**, also known as a **full outer join**, is a type of join operation in relational databases that combines the results of a **left join** and a **right join**. It returns all records from both tables involved in the join, matching rows where there is a common key and including unmatched rows from both tables. For rows that do not have a match in the other table, the result will contain NULL values in the corresponding columns. This join is particularly useful when you need a comprehensive view of two datasets, ensuring that no data from either table is excluded, even if there is no relationship between them.



# Cross Join:

A **cross join** is a type of join in relational databases that returns the Cartesian product of two tables. This means every row from the first table is paired with every row from the second table, resulting in a number of rows equal to the product of the row counts of both tables. Unlike other types of joins, a cross join does not require any condition or relationship between the tables. It is often used for scenarios where all possible combinations of rows are needed, such as generating test data or combining datasets without predefined relationships.

SELECT Fruits.FruitName, Colors.ColorName

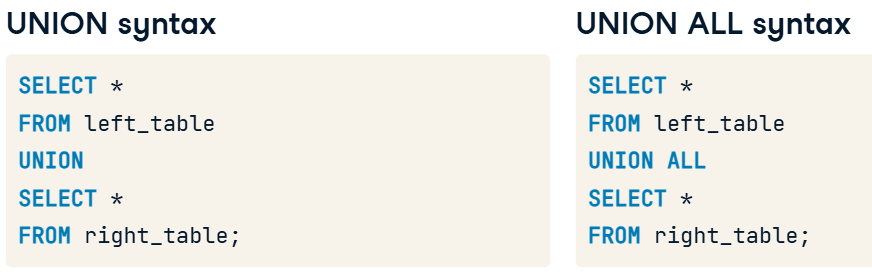
FROM Fruits

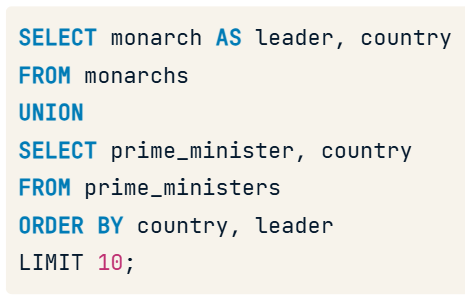
CROSS JOIN Colors;

# Self-Join:

# Union and Union All:

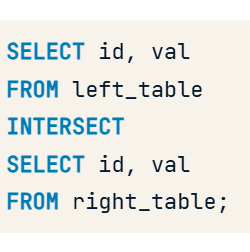
In SQL, **UNION** and **UNION ALL** are used to combine the results of two or more SELECT queries into a single result set. The key difference lies in how they handle duplicate records. The **UNION** operator eliminates duplicate rows from the result set, ensuring that only distinct rows are included. In contrast, **UNION ALL** includes all rows from the combined queries, including duplicates, which makes it faster and more efficient since it doesn't require the database to check for duplicates. Both operators require the number of columns and their data types in the SELECT queries to match, and the order of the columns in each query should align.





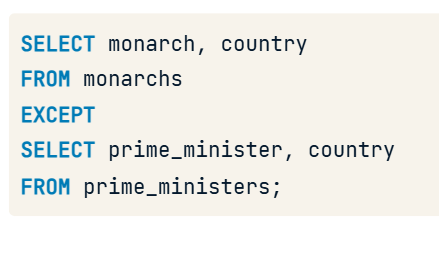
# Intersect:

In SQL, the **INTERSECT** operator is used to return the common rows that exist in the result sets of two SELECT queries. It essentially finds the intersection of the datasets, including only those rows that are present in both queries. Like **UNION**, the **INTERSECT** operator automatically removes duplicate rows from the result, ensuring that the output consists of unique rows. For **INTERSECT** to work, the number of columns and their data types in the SELECT queries must match, and the order of the columns should align. This operator is useful for identifying overlapping data between tables or queries.



# Except:

EXCEPT allows us to identify the records that are present in one table, but not the other. More specifically, it retains only records from the left table that are not present in the right table.



# Semi Join:

A **semi-join** in SQL refers to a query where you are interested in finding rows from one table that have a matching row in another table, but you do **not** retrieve columns from the second table. Essentially, it's like a filtered result based on the existence of related data in another table.

Semi-joins are often implemented using the **EXISTS** or **IN** clauses.

**Key Characteristics of a Semi-Join:**

* It checks for the existence of a match in the other table.
* Only returns columns from the first table.
* Unlike an **INNER JOIN**, it doesn’t combine data from both tables.

Anti-Join:

An **anti-join** in SQL is the opposite of a **semi-join**. It is used to find rows in one table that **do not** have matching rows in another table. Like the semi-join, the anti-join is a **concept**, not a specific SQL keyword, and it is typically implemented using subqueries with NOT EXISTS or NOT IN.

**Key Characteristics of an Anti-Join:**

* It returns rows from the first table where no corresponding match exists in the second table.
* Like the semi-join, it does not retrieve columns from the second table.
* Commonly used to identify unmatched or "orphaned" rows.