SQL is a standard language for accessing and manipulating databases.

What is SQL?

* SQL stands for Structured Query Language
* SQL lets you access and manipulate databases
* SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987

What Can SQL do?

* SQL can execute queries against a database
* SQL can retrieve data from a database
* SQL can insert records in a database
* SQL can update records in a database
* SQL can delete records from a database
* SQL can create new databases
* SQL can create new tables in a database
* SQL can create stored procedures in a database
* SQL can create views in a database
* SQL can set permissions on tables, procedures, and views

SQL is a Standard - BUT....

Although SQL is an ANSI/ISO standard, there are different versions of the SQL language.

However, to be compliant with the ANSI standard, they all support at least the major commands (such as SELECT, UPDATE, DELETE, INSERT, WHERE) in a similar manner.

Note: Most of the SQL database programs also have their own proprietary extensions in addition to the SQL standard!

Using SQL in Your Web Site

To build a web site that shows data from a database, you will need:

* An RDBMS database program (i.e. MS Access, SQL Server, MySQL)
* To use a server-side scripting language, like PHP or ASP
* To use SQL to get the data you want
* To use HTML / CSS to style the page

RDBMS

RDBMS stands for Relational Database Management System.

RDBMS is the basis for SQL, and for all modern database systems such as MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.

The data in RDBMS is stored in database objects called tables. A table is a collection of related data entries and it consists of columns and rows.

Look at the "Customers" table:

Example

SELECT \* FROM Customers;

Every table is broken up into smaller entities called fields. The fields in the Customers table consist of CustomerID, CustomerName, ContactName, Address, City, PostalCode and Country. A field is a column in a table that is designed to maintain specific information about every record in the table.

A record, also called a row, is each individual entry that exists in a table. For Example, there are 91 records in the above Customers table. A record is a horizontal entity in a table.

A column is a vertical entity in a table that contains all information associated with a specific field in a table.

Database Tables

A database most often contains one or more tables. Each table is identified by a name (e.g. "Customers" or "Orders"). Tables contain records (rows) with data.

In this tutorial we will use the well-known Northwind sample database (included in MS Access and MS SQL Server).

Below is a selection from the "Customers" table:

The table above contains five records (one for each customer) and seven columns (CustomerID, CustomerName, ContactName, Address, City, PostalCode, and Country).

SQL Statements

Most of the actions you need to perform on a database are done with SQL statements.

The following SQL statement selects all the records in the "Customers" table:

Example

SELECT \* FROM Customers;

Keep in Mind That...

* SQL keywords are NOT case sensitive: select is the same as SELECT

In this tutorial we will write all SQL keywords in upper-case.

SEMICOLON AFTER SQL STATEMENTS?

Some database systems require a semicolon at the end of each SQL statement.

Semicolon is the standard way to separate each SQL statement in database systems that allow more than one SQL statement to be executed in the same call to the server.

In this tutorial, we will use semicolon at the end of each SQL statement.

Some of The Most Important SQL Commands

* SELECT - extracts data from a database
* UPDATE - updates data in a database
* DELETE - deletes data from a database
* INSERT INTO - inserts new data into a database
* CREATE DATABASE - creates a new database
* ALTER DATABASE - modifies a database
* CREATE TABLE - creates a new table
* ALTER TABLE - modifies a table
* DROP TABLE - deletes a table
* CREATE INDEX - creates an index (search key)
* DROP INDEX - deletes an index

The SQL SELECT Statement

The SELECT statement is used to select data from a database.

The data returned is stored in a result table, called the result-set.

SELECT Syntax

SELECT column1, column2, ...  
FROM table\_name;

Here, column1, column2, ... are the field names of the table you want to select data from. If you want to select all the fields available in the table, use the following syntax:

SELECT \* FROM table\_name;

Demo Database

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

SELECT Column Example

The following SQL statement selects the "CustomerName" and "City" columns from the "Customers" table:

Example

SELECT CustomerName, City FROM Customers;

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_columns)

SELECT \* Example

The following SQL statement selects all the columns from the "Customers" table:

Example

SELECT \* FROM Customers;

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_all)

The SQL SELECT DISTINCT Statement

The SELECT DISTINCT statement is used to return only distinct (different) values.

Inside a table, a column often contains many duplicate values; and sometimes you only want to list the different (distinct) values.

The SELECT DISTINCT statement is used to return only distinct (different) values.

SELECT DISTINCT Syntax

SELECT DISTINCT column1, column2, ...  
FROM table\_name;

SELECT Example

The following SQL statement selects all (and duplicate) values from the "Country" column in the "Customers" table:

Example

SELECT Country FROM Customers;

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_no_distinct)

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

SELECT DISTINCT Examples

The following SQL statement selects only the DISTINCT values from the "Country" column in the "Customers" table:

Example

SELECT DISTINCT Country FROM Customers;

The following SQL statement lists the number of different (distinct) customer countries:

Example

SELECT COUNT(DISTINCT Country) FROM Customers;

Note: The Example above will not work in Firefox and Microsoft Edge! Because COUNT(DISTINCT column\_name) is not supported in Microsoft Access databases. Firefox and Microsoft Edge are using Microsoft Access in our Examples.

Here is the workaround for MS Access:

Example

SELECT Count(\*) AS DistinctCountries  
FROM (SELECT DISTINCT Country FROM Customers);

The SQL WHERE Clause

The WHERE clause is used to filter records.

The WHERE clause is used to extract only those records that fulfill a specified condition.

WHERE Syntax

SELECT column1, column2, ...  
FROM table\_name  
WHERE condition;

Note: The WHERE clause is not only used in SELECT statement, it is also used in UPDATE, DELETE statement, etc.!

Demo Database

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

WHERE Clause Example

The following SQL statement selects all the customers from the country "Mexico", in the "Customers" table:

Example

SELECT \* FROM Customers  
WHERE Country='Mexico';

Text Fields vs. Numeric Fields

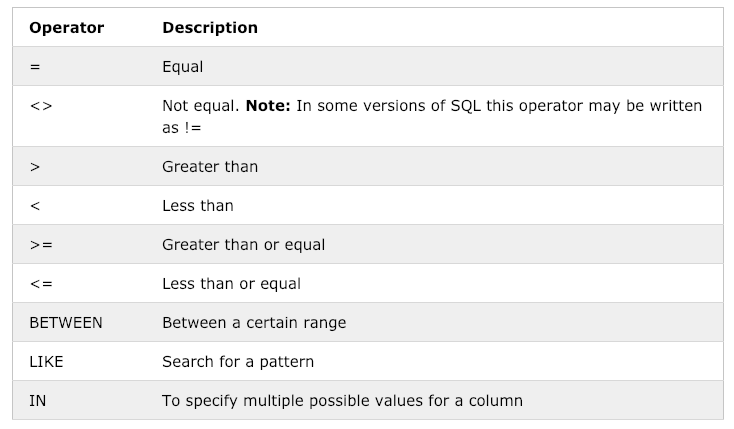
SQL requires single quotes around text values (most database systems will also allow double quotes).

However, numeric fields should not be enclosed in quotes:

Example

SELECT \* FROM Customers  
WHERE CustomerID=1;

**Operators in The WHERE Clause. The following operators can be used in the WHERE clause:**



**The SQL AND, OR and NOT Operators**

The WHERE clause can be combined with AND, OR, and NOT operators.

The AND and OR operators are used to filter records based on more than one condition:

* The AND operator displays a record if all the conditions separated by AND are TRUE.
* The OR operator displays a record if any of the conditions separated by OR is TRUE.

The NOT operator displays a record if the condition(s) is NOT TRUE.

**Logical Operator Precedence:** An SQL rule stating that in the execution of a query, the operator AND is applied first, while OR is applied second. SQL starts executing the AND operator first regardless of position.

AND Syntax – Used for conditions set on different columns

SELECT column1, column2, ...  
FROM table\_name  
WHERE condition1 AND condition2 AND condition3 ...;

AND Example

The following SQL statement selects all fields from "Customers" where country is "Germany" AND city is "Berlin":

Example

SELECT \* FROM Customers  
WHERE Country='Germany' AND City='Berlin';

OR Syntax - Used for conditions set on the same columns

SELECT column1, column2, ...  
FROM table\_name  
WHERE condition1 OR condition2 OR condition3 ...;

OR Example

The following SQL statement selects all fields from "Customers" where city is "Berlin" OR "München":

Example

SELECT \* FROM Customers  
WHERE City='Berlin' OR City='München';

NOT Syntax

SELECT column1, column2, ...  
FROM table\_name  
WHERE NOT condition;

NOT Example

The following SQL statement selects all fields from "Customers" where country is NOT "Germany":

Example

SELECT \* FROM Customers  
WHERE NOT Country='Germany';

**Combining AND, OR and NOT**

You can also combine the AND, OR and NOT operators.

The following SQL statement selects all fields from "Customers" where country is "Germany" AND city must be "Berlin" OR "München" (use parenthesis to form complex expressions):

Example

SELECT \* FROM Customers  
WHERE Country='Germany' AND (City='Berlin' OR City='München');

The following SQL statement selects all fields from "Customers" where country is NOT "Germany" and NOT "USA":

Example

SELECT \* FROM Customers  
WHERE NOT Country='Germany' AND NOT Country='USA';

**The SQL ORDER BY Keyword**

The ORDER BY keyword is used to sort the result-set in ascending or descending order.

The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

ORDER BY Syntax

SELECT *column1*,*column2, ...*  
FROM *table\_name*  
ORDER BY *column1, column2, ...*ASC|DESC;

ORDER BY Example

The following SQL statement selects all customers from the "Customers" table, sorted by the "Country" column:

Example

SELECT \* FROM Customers  
ORDER BY Country;

ORDER BY DESC Example

The following SQL statement selects all customers from the "Customers" table, sorted DESCENDING by the "Country" column:

Example

SELECT \* FROM Customers  
ORDER BY Country DESC;

ORDER BY Several Columns Example

The following SQL statement selects all customers from the "Customers" table, sorted by the "Country" and the "CustomerName" column. This means that it orders by Country, but if some rows have the same Country, it orders them by CustomerName:

Example

SELECT \* FROM Customers  
ORDER BY Country, CustomerName;

ORDER BY Several Columns Example 2

The following SQL statement selects all customers from the "Customers" table, sorted ascending by the "Country" and descending by the "CustomerName" column:

Example

SELECT \* FROM Customers  
ORDER BY Country ASC, CustomerName DESC;

**GROUP BY:**

Results may be grouped according to a specific field or fields. It is a powerful and useful tool SQL.

GROUP BY must be placed immediately after the WHERE conditions, if any, and just before the ORDER BY clause. **Only distinct values will be selected.**

**Note**: It is useful when using aggregate functions. Always include the field that you have grouped your results by in the select statement to ensure they are reflected in the results.

**Aggregate Functions:** They gather data from many rows of a table, then aggregate it into a *single* value. Types include:

- CONUNT(), COUNT(DISTINCT()

- SUM()

- MIN()

- MAX()

- AVG()

You can use ROUND() for numerical decimal places. Example ROUND(AVG(Salary),2) – this yields (\_\_.xx). Putting no number yields a solid figure.

Example: 1

SELECT *column1*,*column2, ...*  
FROM *table\_name*

WHERE *conditions*

GROUP BY *column1*,*column2, ...*  
ORDER BY *column1, column2, ...*ASC|DESC;

Example: 2

SELECT *column1*, COUNT(*column1)*  
FROM *table\_name*

GROUP BY *column1*

ORDER BY *column1 or column2, ...*ASC|DESC;

Example: 3 Using an alias

SELECT *column1*, COUNT(*column1) as XXX*  
FROM *table\_name*

GROUP BY *column1*

ORDER BY *XXX, ...*ASC|DESC;

**The SQL HAVING Clause (similar to WHERE but different abilities)**

Refines the output from records that do not satisfy a certain condition. It is frequently implemented with GROUP BY. HAVING can be applied to subsets for aggerated groups.

After HAVING, you can have a condition with an **aggregate function,** while WHERE cannot use aggregate functions within its conditions.

HAVING is like WHERE but applied to the GROUP BY block. Although HAVING works for both aggregate and non-aggregate functions, you cannot run both at the same time e.g. HAVING xxxx AND xxxx.

* Aggregate functions: GROUP BY, HAVING
* General functions: WHERE

Example

SELECT Country, count (Country) FROM Customers  
GROUP BY Country

HAVING count (Country) >10

Order by Country asc;

Example

SELECT first\_name, count (first\_name) FROM Customers

WHERE start\_date > ‘1999-01-01’  
GROUP BY first\_name

HAVING count (first\_name) >10

Order by first\_name asc;

**The SQL INSERT INTO Statement**

The INSERT INTO statement is used to insert new records in a table.

INSERT INTO Syntax

It is possible to write the INSERT INTO statement in two ways.

The first way specifies both the column names and the values to be inserted:

INSERT INTO *table\_name* (*column1*,*column2*,*column3*, ...)  
VALUES (*value1*,*value2*,*value3*, ...);

If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table. The INSERT INTO syntax would be as follows:

INSERT INTO *table\_name*  
VALUES (*value1*,*value2*,*value3*, ...);

INSERT INTO Example

The following SQL statement inserts a new record in the "Customers" table:

Example

INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)  
VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway');

**What is a NULL Value?**

A field with a NULL value is a field with no value.

If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value.

How to Test for NULL Values?

It is not possible to test for NULL values with comparison operators, such as =, <, or <>.

We will have to use the IS NULL and IS NOT NULL operators instead.

IS NULL Syntax

SELECT *column\_names*FROM *table\_name*  
WHERE *column\_name* IS NULL;

IS NOT NULL Syntax

SELECT *column\_names*FROM *table\_name*  
WHERE *column\_name* IS NOT NULL;

The IS NULL Operator

The IS NULL operator is used to test for empty values (NULL values).

The following SQL lists all customers with a NULL value in the "Address" field:

Example

SELECT CustomerName, ContactName, Address  
FROM Customers  
WHERE Address IS NULL;

**Tip:** Always use IS NULL to look for NULL values.

The IS NOT NULL Operator

The IS NOT NULL operator is used to test for non-empty values (NOT NULL values).

The following SQL lists all customers with a value in the "Address" field:

Example

SELECT CustomerName, ContactName, Address  
FROM Customers  
WHERE Address IS NOT NULL;

Top of Form

**The SQL UPDATE Statement**

The UPDATE statement is used to modify the existing records in a table.

UPDATE Syntax

UPDATE *table\_name*  
SET *column1*=*value1*,*column2*=*value2*, ...  
WHERE *condition*;

UPDATE Table

The following SQL statement updates the first customer (CustomerID = 1) with a new contact person *and* a new city.

Example

UPDATE Customers  
SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'  
WHERE CustomerID = 1;

UPDATE Multiple Records

It is the WHERE clause that determines how many records that will be updated.

The following SQL statement will update the contactname to "Juan" for all records where country is "Mexico":

Example

UPDATE Customers  
SET ContactName='Juan'  
WHERE Country='Mexico';

Bottom of Form

Update Warning!

Be careful when updating records. If you omit the WHERE clause, ALL records will be updated!

Example

UPDATE Customers  
SET ContactName='Juan';

**The SQL DELETE Statement**

The DELETE statement is used to delete existing records in a table.

DELETE Syntax

DELETE FROM *table\_name*WHERE *condition*;

**Note:** Be careful when deleting records in a table! Notice the WHERE clause in the DELETE statement. The WHERE clause specifies which record(s) should be deleted. If you omit the WHERE clause, all records in the table will be deleted!

SQL DELETE Example

The following SQL statement deletes the customer "Alfreds Futterkiste" from the "Customers" table:

Example

DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste';

Delete All Records

It is possible to delete all rows in a table without deleting the table. This means that the table structure, attributes, and indexes will be intact:

DELETE FROM *table\_name*;

The following SQL statement deletes all rows in the "Customers" table, without deleting the table:

Example

DELETE FROM Customers;

**The SQL SELECT TOP Clause**

The SELECT TOP clause is used to specify the number of records to return.

The SELECT TOP clause is useful on large tables with thousands of records. Returning a large number of records can impact on performance.

**Note:** Not all database systems support the SELECT TOP clause. MySQL supports the LIMIT clause to select a limited number of records, while Oracle uses ROWNUM.

**SQL Server / MS Access Syntax:**

SELECT TOP *number*|*percent* *column\_name(s)*  
FROM *table\_name*WHERE *condition*;

**MySQL Syntax:**

SELECT *column\_name(s)*  
FROM *table\_name*WHERE *condition*  
LIMIT *number*;

**Oracle Syntax:**

SELECT *column\_name(s)*  
FROM *table\_name*  
WHERE ROWNUM <= *number*;

**SQL TOP, LIMIT and ROWNUM Examples**

The following SQL statement selects the first three records from the "Customers" table:

Example

SELECT TOP 3 \* FROM Customers;

The following SQL statement shows the equivalent Example using the LIMIT clause: Add at the end of the code just before the semi colon e.g. Limit 3.

Example

SELECT \* FROM Customers  
LIMIT 3;

The following SQL statement shows the equivalent Example using ROWNUM:

Example

SELECT \* FROM Customers  
WHERE ROWNUM <= 3;

**SQL TOP PERCENT Example**

The following SQL statement selects the first 50% of the records from the "Customers" table:

Example

SELECT TOP 50 PERCENT \* FROM Customers;

ADD a WHERE CLAUSE

The following SQL statement selects the first three records from the "Customers" table, where the country is "Germany":

Example

SELECT TOP 3 \* FROM Customers  
WHERE Country='Germany';

The following SQL statement shows the equivalent Example using the LIMIT clause:

Example

SELECT \* FROM Customers  
WHERE Country='Germany'  
LIMIT 3;

The following SQL statement shows the equivalent Example using ROWNUM:

Example

SELECT \* FROM Customers  
WHERE Country='Germany' AND ROWNUM <= 3;

**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*;

MAX() Syntax

SELECT MAX(*column\_name*)  
FROM *table\_name*  
WHERE *condition*;

MIN() Example

The following SQL statement finds the price of the cheapest product:

Example

SELECT MIN(Price) AS SmallestPrice  
FROM Products;

MAX() Example

The following SQL statement finds the price of the most expensive product:

Example

SELECT MAX(Price) AS LargestPrice  
FROM Products;

**SQL Aggregate Functions: COUNT(), AVG() and SUM() Functions**

The COUNT() function returns the number of rows that matches a specified criteria.

The AVG() function returns the average value of a numeric column.

The SUM() function returns the total sum of a numeric column.

**Note: They ignore NULL values unless told not to**

COUNT() Syntax: Parenthesis after COUNT() must begin right after and not after whitespace

SELECT COUNT(*column\_name*):   
FROM *table\_name*  
WHERE *condition*;

AVG() Syntax

SELECT AVG(*column\_name*)  
FROM *table\_name*  
WHERE *condition*;

SUM() Syntax

SELECT SUM(*column\_name*)  
FROM *table\_name*  
WHERE *condition*;

COUNT() Example

The following SQL statement finds the number of products:

Example

SELECT COUNT(ProductID)  
FROM Products;

**Note:** NULL values are not counted.

AVG() Example

The following SQL statement finds the average price of all products:

Example

SELECT AVG(Price)  
FROM Products;

**Note:** NULL values are ignored.

SUM() Example

The following SQL statement finds the sum of the "Quantity" fields in the "OrderDetails" table:

Example

SELECT SUM(Quantity)  
FROM OrderDetails;

**The SQL LIKE Operator**

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

There are two wildcards used in conjunction with the LIKE operator:

* % - The percent sign represents zero, one, or multiple characters
* \_ - The underscore represents a single character

**Note:** MS Access uses a question mark (?) instead of the underscore (\_).

The percent sign and the underscore can also be used in combinations!

LIKE Syntax

SELECT *column1, column2, ...*  
FROM *table\_name*  
WHERE *columnN* LIKE *pattern*;

**Tip:** You can also combine any number of conditions using AND or OR operators.

Here are some Examples showing different LIKE operators with '%' and '\_' wildcards:



SQL LIKE Examples

The following SQL statement selects all customers with a CustomerName starting with "a":

Example

SELECT \* FROM Customers  
WHERE CustomerName LIKE 'a%';

The following SQL statement selects all customers with a CustomerName ending with "a":

Example

SELECT \* FROM Customers  
WHERE CustomerName LIKE '%a';

The following SQL statement selects all customers with a CustomerName that have "or" in any position:

Example

SELECT \* FROM Customers  
WHERE CustomerName LIKE '%or%';

The following SQL statement selects all customers with a CustomerName that have "r" in the second position:

Example

SELECT \* FROM Customers  
WHERE CustomerName LIKE '\_r%';

The following SQL statement selects all customers with a CustomerName that starts with "a" and are at least 3 characters in length:

Example

SELECT \* FROM Customers  
WHERE CustomerName LIKE 'a\_%\_%';

The following SQL statement selects all customers with a ContactName that starts with "a" and ends with "o":

Example

SELECT \* FROM Customers  
WHERE ContactName LIKE 'a%o';

The following SQL statement selects all customers with a CustomerName that does NOT start with "a":

Example

SELECT \* FROM Customers  
WHERE CustomerName NOT LIKE 'a%';

**SQL Wildcard Characters**

A wildcard character is used to substitute any other character(s) in a string.

Wildcard characters are used with the [SQL LIKE](https://www.w3schools.com/sql/sql_like.asp) operator. The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

There are two wildcards used in conjunction with the LIKE operator:

* % - The percent sign represents zero, one, or multiple characters
* \_ - The underscore represents a single character

**Note:** MS Access uses a question mark (?) instead of the underscore (\_).

In MS Access and SQL Server, you can also use:

* [*charlist*] - Defines sets and ranges of characters to match
* [^*charlist*] or [!*charlist*] - Defines sets and ranges of characters NOT to match

The wildcards can also be used in combinations!

Using the % Wildcard

The following SQL statement selects all customers with a City starting with "ber":

Example

SELECT \* FROM Customers

WHERE City LIKE 'ber%';

The following SQL statement selects all customers with a City containing the pattern "es":

Example

SELECT \* FROM Customers

WHERE City LIKE '%es%';

Using the \_ Wildcard

The following SQL statement selects all customers with a City starting with any character, followed by "erlin":

Example

SELECT \* FROM Customers

WHERE City LIKE '\_erlin';

The following SQL statement selects all customers with a City starting with "L", followed by any character, followed by "n", followed by any character, followed by "on":

Example

SELECT \* FROM Customers

WHERE City LIKE 'L\_n\_on';

Using the [charlist] Wildcard

The following SQL statement selects all customers with a City starting with "b", "s", or "p":

Example

SELECT \* FROM Customers

WHERE City LIKE '[bsp]%';

The following SQL statement selects all customers with a City starting with "a", "b", or "c":

Example

SELECT \* FROM Customers

WHERE City LIKE '[a-c]%';

Using the [!charlist] Wildcard

The two following SQL statements select all customers with a City NOT starting with "b", "s", or "p":

Example

SELECT \* FROM Customers

WHERE City LIKE '[!bsp]%';

Or:

Example

SELECT \* FROM Customers

WHERE City NOT LIKE '[bsp]%';

**The SQL 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);

IN Operator Examples

The following SQL statement selects all customers that are located in "Germany", "France" and "UK":

Example

SELECT \* FROM Customers

WHERE Country IN ('Germany', 'France', 'UK');

The following SQL statement selects all customers that are NOT located in "Germany", "France" or "UK":

Example

SELECT \* FROM Customers

WHERE Country NOT IN ('Germany', 'France', 'UK');

The following SQL statement selects all customers that are from the same countries as the suppliers:

Example

SELECT \* FROM Customers

WHERE Country IN (SELECT Country FROM Suppliers);

The following SQL statement selects all customers in FRANCE that have suppliers in FRANCE

Example

SELECT \* FROM Customers

WHERE Country IN (SELECT Country FROM Suppliers Where country LIKE 'FRANCE');

The following SQL statement selects all customers where CustomerName FRANCE with suppliers in FRANCE

Example

SELECT \* FROM Customers WHERE CustomerName IN (SELECT Country FROM Suppliers Where country LIKE 'FRANCE')

**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;

​

BETWEEN Example

The following SQL statement selects all products with a price BETWEEN 10 and 20:

Example

SELECT \* FROM Products  
WHERE Price BETWEEN 10 AND 20;

NOT BETWEEN Example

To display the products outside the range of the previous Example, use NOT BETWEEN:

Example

SELECT \* FROM Products

WHERE Price NOT BETWEEN 10 AND 20;

BETWEEN with IN Example

The following SQL statement selects all products with a price BETWEEN 10 and 20. In addition; do not show products with a CategoryID of 1,2, or 3:

Example

SELECT \* FROM Products

WHERE (Price BETWEEN 10 AND 20)

AND NOT CategoryID IN (1,2,3);

The following SQL statement selects all products with a price BETWEEN 10 and 20. In addition; only show products with a ProductName starting with ‘ch’

SELECT \* FROM Products

WHERE (Price BETWEEN 10 AND 20)

AND ProductName IN (SELECT ProductName from Products where ProductName Like 'ch%');

BETWEEN Text Values Example

The following SQL statement selects all products with a ProductName BETWEEN 'Carnarvon Tigers' and 'Mozzarella di Giovanni':

Example

SELECT \* FROM Products

WHERE ProductName BETWEEN 'Carnarvon Tigers' AND 'Mozzarella di Giovanni'

ORDER BY ProductName;

NOT BETWEEN Text Values Example

The following SQL statement selects all products with a ProductName NOT BETWEEN 'Carnarvon Tigers' and 'Mozzarella di Giovanni':

Example

SELECT \* FROM Products

WHERE ProductName NOT BETWEEN 'Carnarvon Tigers' AND 'Mozzarella di Giovanni'

ORDER BY ProductName;

BETWEEN Dates Example

The following SQL statement selects all orders with an OrderDate BETWEEN '01-July-1996' and '31-July-1996':

Example

SELECT \* FROM Orders

WHERE OrderDate BETWEEN #01/07/1996# AND #31/07/1996#;

Example

SELECT \* FROM Orders

WHERE OrderDate BETWEEN '1996-07-01' AND '1996-07-31';

**SQL Aliases**

SQL aliases are used to give a table, or a column in a table, a temporary name.

Aliases are often used to make column names more readable.

An alias only exists for the duration of the query.

Alias Column Syntax

SELECT column\_name AS alias\_name

FROM table\_name;

Alias Table Syntax

SELECT column\_name(s)

FROM table\_name AS alias\_name;

Alias for Columns Examples

The following SQL statement creates two aliases, one for the CustomerID column and one for the CustomerName column:

Example

SELECT CustomerID AS ID, CustomerName AS Customer

FROM Customers;

The following SQL statement creates two aliases, one for the CustomerName column and one for the ContactName column. Note: It requires double quotation marks or square brackets if the alias name contains spaces:

Example

SELECT CustomerName AS Customer, ContactName AS [Contact Person]

FROM Customers;

The following SQL statement creates an alias named "Address" that combine four columns (Address, PostalCode, City and Country):

Example

SELECT CustomerName, Address + ', ' + PostalCode + ' ' + City + ', ' + Country AS Address

FROM Customers;

Note: To get the SQL statement above to work in MySQL use the following:

SELECT CustomerName, CONCAT(Address,', ',PostalCode,', ',City,', ',Country) AS Address

FROM Customers;

Alias for Tables Example

The following SQL statement selects all the orders from the customer with CustomerID=4 (Around the Horn). We use the "Customers" and "Orders" tables, and give them the table aliases of "c" and "o" respectively (Here we use aliases to make the SQL shorter):

Example

SELECT o.OrderID, o.OrderDate, c.CustomerName

FROM Customers AS c, Orders AS o

WHERE c.CustomerName="Around the Horn" AND c.CustomerID=o.CustomerID;

The following SQL statement is the same as above, but without aliases:

Example

SELECT Orders.OrderID, Orders.OrderDate, Customers.CustomerName

FROM Customers, Orders

WHERE Customers.CustomerName="Around the Horn" AND Customers.CustomerID=Orders.CustomerID;

ALIASES CAN BE USEFUL WHEN:

* There are more than one table involved in a query
* Functions are used in the query
* Column names are big or not very readable
* Two or more columns are combined together

**SQL JOIN**

A JOIN clause is used to combine rows from two or more tables, based on a related column between them. It shows a result set, containing fields derived from two or more tables. The matching columns do not need to have the same name

Using JOINS,

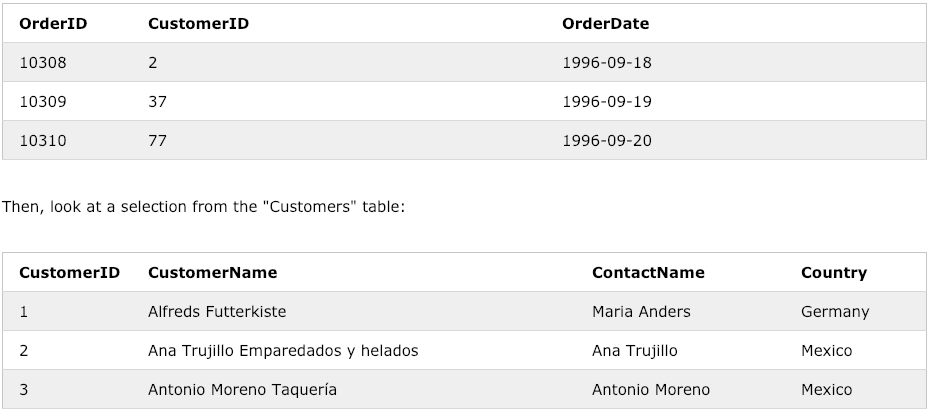
* Find a related column from the two tables that contains the same type of data
* Feel free to add columns from these two tables to the output.
* The columns used to relate tables must represent the same object, such as ID.
* The tables being considered do not need to be logically adjacent in the schema.

**Relational schemas** are the perfect tool that will help find a strategy for linking tables.

A screenshot of a cell phone

Description automatically generated

Let's look at a selection from the "Orders" table:



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

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

Example

SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate 🡨(What is displayed)

FROM Orders

INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID; 🡨(Condition for join)



If using an Alias:

SELECT o.OrderID, c.CustomerName, o.OrderDate

FROM Orders AS o

INNER JOIN Customers as c

ON o.CustomerID = c.CustomerID;

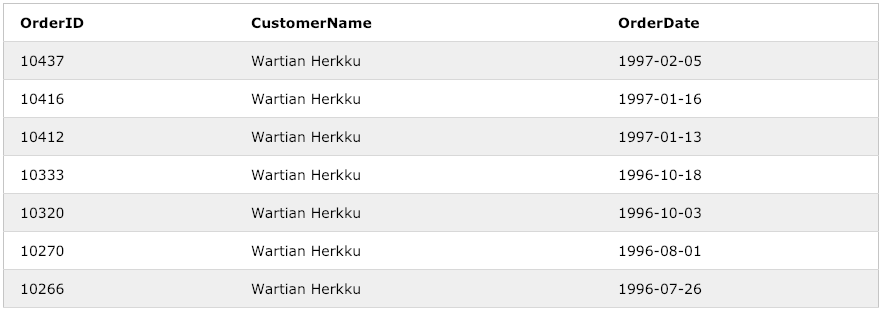
Alias plus where CustomerName starts with ‘Wa’ ordered by OrderDate descending

SELECT o.OrderID, c.CustomerName, o.OrderDate

FROM Orders AS o

INNER JOIN Customers as c

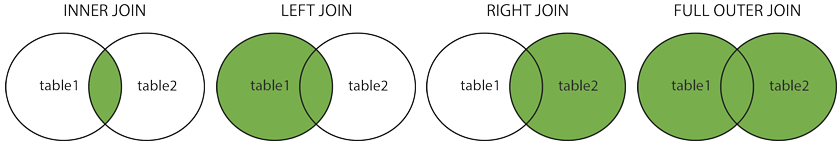
ON o.CustomerID=c.CustomerID where c.customername like 'Wa%' order by o.OrderDate desc



Different Types of SQL JOINs

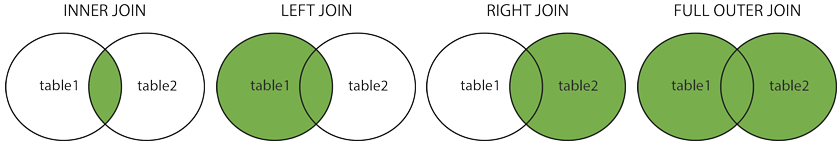
Here are the different types of the JOINs in SQL:

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



**SQL INNER JOIN Keyword**

The INNER JOIN keyword selects records that have matching values or records in both tables. The merging area is called the result set. Inner joins extract only records in which the values in the related columns match. Null values and values appearing in just one of the two tables and not appearing in the other, are not displayed.



INNER JOIN Syntax – JOIN (preferred) also works as INNER JOIN

SELECT table1.column\_name, table2.column\_name etc.

FROM table1

INNER JOIN table2 ON table1.column\_name = table2.column\_name;

SQL INNER JOIN Example

The following SQL statement selects all orders with customer information:

Example

SELECT Orders.OrderID, Customers.CustomerName

FROM Orders

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

Note: The INNER JOIN keyword selects all rows from both tables as long as there is a match between the columns. If there are records in the "Orders" table that do not have matches in "Customers", these orders will not be shown!

DUPLICATE RECORDS

Also known as duplicate rows, are identical rows in an SQL table. For a pair of duplicate records, the value in each column coincide. Data sets should be cleaned to avoid them, sometimes they may be present in raw data.

To avoid duplicate columns in a result, attach the GROUP BY function using the most distinct column e.g. ID.

Example

SELECT table1.column\_name, table2.column\_name etc.

FROM table1

INNER JOIN table2 ON table1.column\_name = table2.column\_name

GROUP BY table1.column\_name

ORDER BY table1.column\_name;

JOIN Three Tables

The following SQL statement selects all orders with customer and shipper information:

Example

SELECT Orders.OrderID, Customers.CustomerName, Shippers.ShipperName

FROM ((Orders

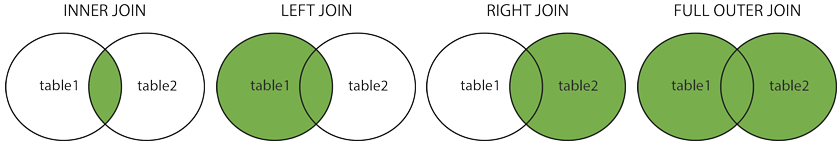
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID)

INNER JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID);

SQL LEFT JOIN Keyword

This returns all records from the left table (table1), and the matched records from the right table (table2). The result is NULL from the right side, if there is no match. In contrast to INNER JOIN, when working with LEFT JOINs, the order in which you join tables **matters**.

Values from the right table will be included only if their linking column contains a value coinciding or matching with a value from the linking column of the left table.



LEFT JOIN Syntax

SELECT table1.column\_name, table2.column\_name etc.

FROM table1

INNER JOIN table2 ON table1.column\_name = table2.column\_name;

Note:

* Retrieve the first selection from the first table selected in the join syntax.
* In some databases LEFT JOIN is called LEFT OUTER JOIN.
* The LEFT JOIN keyword returns all records from the left table (Customers), even if there are no matches in the right table (Orders).

SQL LEFT JOIN Example

The following SQL statement will select all customers, and any orders they might have:

Example

SELECT Customers.CustomerName, Orders.OrderID

FROM Customers

LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID

ORDER BY Customers.CustomerName;

Example 2

SELECT Customers.CustomerName, Orders.OrderID

FROM Customers

LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID

WHERE OrderID = ‘-----'

ORDER BY Customers.CustomerName;

USING JOIN + WHERE

The JOIN keyword is used for connecting the respective tables while WHERE is used to define the condition(s) that will determine which will be connecting point between the two tables.

Example

SELECT Customers.CustomerName, Orders.OrderID

FROM Customers

LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID

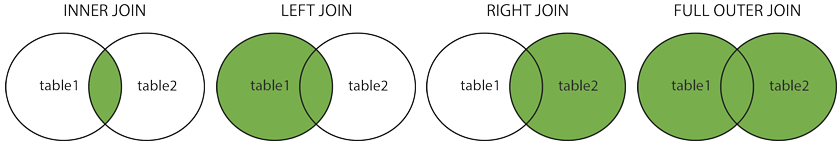
WHERE OrderID = ‘\_ \_ \_ \_'

ORDER BY Customers.CustomerName;

SQL RIGHT JOIN Keyword

This returns all records from the right table (table2), and the matched records from the left table (table1). The result is NULL from the left side, when there is no match.

Values from the left table will be included only if their linking column contains a value coinciding or matching with a value from the linking column of the right table.



RIGHT JOIN Syntax

SELECT table1.column\_name, table2.column\_name etc.

FROM table1

INNER JOIN table2 ON table1.column\_name = table2.column\_name;

Note:

* In some databases RIGHT JOIN is called RIGHT OUTER JOIN.
* Switching table placements and LEFT/RIGHT JOIN keywords accordingly will yield the same results.
* RIGHT JOIN is seldom used in practice
* Linking column is the same as the matching column
* LEFT and RIGHT joins are perfect examples of one-to-many relationships.

SQL RIGHT JOIN Example

The following SQL statement will return all employees, and any orders they might have placed:

Example

SELECT Orders.OrderID, Employees.LastName, Employees.FirstName

FROM Orders

RIGHT JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID

ORDER BY Orders.OrderID;

SQL CROSS JOIN Keyword

This will take the values from a certain table and connect them with all the values from the tables we want to join it with. It will connect all the values, not just those that match. From a mathematical standpoint, a CROSS JOIN is the cartesian product of the values of two or more sets. It does not require any conditions to join tables.

Example

SELECT table1. \*, table2. \*

FROM table1

CROSS JOIN table2

ORDER BY table1.column\_name, table2.column\_name;

Example

SELECT table1. \*, table2. \*

FROM table1

CROSS JOIN table2

WHERE table1.column\_x <> table2.column\_x

ORDER BY table1.column\_name, table2.column\_name;

Note: CROSS JOIN may be used to join multiple tables. However, may potentially return very large result sets!

Example

SELECT table3. \*, table1. \*

FROM table1

CROSS JOIN table2

JOIN table3 ON table2.column\_x = table3.column\_x

WHERE table1.column\_y <> table2.column\_y

ORDER BY table1.column\_name, table2.column\_name;

Note: When creating a query that joins multiple tables, you must back it with strong intuition and a crystal-clear idea of how you would like the tables to be connected.

Example

SELECT table1.column\_name, table2.column\_name, table3.column\_name etc.

FROM table1

JOIN table2 ON table1.column\_x = table2.column\_x

JOIN table3 ON table2.column\_y = table3.column\_y;

SQL FULL OUTER JOIN Keyword

The FULL OUTER JOIN keyword returns all records when there is a match in either left (table1) or right (table2) table records.

A picture containing game, drawing

Description automatically generated

FULL OUTER JOIN Syntax

SELECT column\_name(s)

FROM table1

FULL OUTER JOIN table2 ON table1.column\_name = table2.column\_name;

Note: FULL OUTER JOIN can potentially return very large result-sets!

SQL FULL OUTER JOIN Example

The following SQL statement selects all customers, and all orders:

SELECT Customers.CustomerName, Orders.OrderID

FROM Customers

FULL OUTER JOIN Orders ON Customers.CustomerID=Orders.CustomerID

ORDER BY Customers.CustomerName;

A selection from the result set may look like this:

A screenshot of a cell phone

Description automatically generated

SQL Self JOIN

A self-JOIN is a regular join, but the table is joined with itself.

Self-JOIN Syntax

SELECT column\_name(s)

FROM table1 T1, table1 T2

WHERE condition;

Tips and Tricks for Joins

* Look for key columns which are common between the tables involved in the analysis and are necessary.
* These columns do not need to be foreign or private keys
* Results may be ordered by columns not explicitly stated in the SELECT section.
* When using aggregate functions (AVG, COUNT etc.) ensure to add the GROUP BY function.
* For absolute clarity, always specify where GROUP or ORDER BY columns are sourced from

UNION vs. UNION ALL

Th UNION ALL operator is used to combine a few SELECT statements in a single output. It is a tool that allows the unification of tables.

Note: When uniting two identically organized tables:

* UNION displays only distinct values in the output – requires more computational power and storage.
* UNION ALL retrieves the duplicates as well

Note: Columns selected from each table must:

* Be the same in number
* Have the same name
* Be in the same order
* Contain related data types

Example

SELECT table1.column\_name, table1.column\_name etc.

FROM table1

UNION ALL SELECT table2.column\_name, table2.column\_name etc.

FROM table2;

Example

SELECT table1.column\_name, table1.column\_name etc.

FROM table1

UNION SELECT table2.column\_name, table2.column\_name etc.

FROM table2;

SUB QUERIES OR INNER QUERIES OR NESTED QUERIES **(INNER SELECT)**

These are queries embedded in a query, they are part of another query called an outer query (outer select).

A sub query should always be placed within parenthesis. SQL engines start running the nested or inner query before suing the result to run the outer query. Apply sort keywords like ORDER BY to the other query

Note: A subquery may return a single value (a scalar), a single row, a single column, or an entire table.

* There can exist a lot more than one subquery in your outer query
* It is possible to nest inner queries within other inner queries.
* The SQL engine will execute the innermost query first.

Example

SELECT table1.column\_name, table1.column\_name etc.

FROM table1

WHERE table1.column\_x IN (SELECT table2.column\_x FROM table2) ORDER BY table1.column\_x;

Example

SELECT table1.column\_name, table1.column\_name etc.

FROM table1

WHERE table1.column\_x IN (SELECT table2.column\_x FROM table2

WHERE table2.column\_name BETWEEN ‘\_ \_ \_ \_’ AND ‘\_ \_ \_ \_’);

EXISTS Checks whether certain row values are found within a subquery. This check is conducted row by row and returns a Boolean value (True/False):

* If a row value of a subquery EXISTS, it returns TRUE and the corresponding record of the outer query is extracted.
* If a row value of a subquery doesn’t EXIST, it returns FALSE and no row value from the outer query is extracted.

Example

SELECT table1.column\_name, table1.column\_name etc.

FROM table1

WHERE EXISTS (SELECT \* FROM table2

WHERE table1.column\_x = table2.column\_x)

ORDER BY table1.column\_x;

EXISTS vs. IN

* EXISTS tests row values for existence while IN searches among values
* EXISTS is quicker in retrieving large amounts of data while IN is faster with smaller data sets.

QUERY NOTES: Some nested queries can be rewritten using joins, which are more efficient in general. This is true particularly for inner queries using the WHERE clause. However, subqueries allow for better structuring of the outer query. Thus, each inner query may be thought of in isolation.

Also, in some situations, the use of subqueries is much more intuitive compared to the use of complex joins and unions. It also a lot more structured and readable.

SUB QUERIES IN SELECT AND FROM

Subqueries can be executed within a SELECT statement or with a FROM clause

STORED ROUTINES An SQL statement that can be store on the database server. Whenever a user needs to run the query in question, they can call, reference, or invoke the routine. A stored routine can perform a calculation that transforms an input value in an output value. It can either be a Procedure or a Function.

For instance - A stored routine with an algorithm that:

* Checks all monthly sales generated throughout a calendar year
* Returns the lowest of these values
* The routine can bring the desired result multiple times

**Semi-Colons:**

* They function as a statement terminator.
* Technically, they can also be called delimiters.
* By typing DELIMITER $$, we’ll be able to use the dollar symbols as a delimiter.

Example

DELIMITER $$

CREATE PROCEDURE procedure\_name (param\_1, param\_2)

**Parameters** represent certain values that the procedure will use to complete the calculation it is supposed to execute. Also, note that a **procedure can be created without parameters!** Nevertheless, the parenthesis must always be attached to its name.

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

**Note:**

* To revert the changes, use DELIMITER ; from this moment on, $$ will not act as a delimiter.
* Use the refresh schema button after creating procedures for them to appear in stored procedures.
* The parenthesis, ( ) at the end, indicates that this is a routine.
* Right click on the stored procedures option under schemas to create a procedure using the system.
* Use DROP PROCUDER procedure\_name; to delete procedures from database.
* Also, to drop procedures, right click on the procedure and choose the drop option, then confirm.
* When dropping a nonparameterized procedure, do not write the parentheses at the end.

NON-PARAMETRIC PROCEDURES: Nonparametric statistics are not based on assumptions, that is, the data can be collected from a sample that does not follow a specific distribution.

Example

USE employees;

DROP procedure IF EXISTS select\_salary;

DELIMITER $$

CREATE PROCEDURE select\_salary ()

BEGIN

SELECT \* FROM Salaries

LIMIT 1000;

END$$

DELIMITER ;

CALL select\_salary();

CALL employees.select\_salary();

STORED PROCEDURES Can take an input value and then use it in the query or queries, written in the body of the procedure. This value is represented by the IN parameter.

A picture containing bird, flower

Description automatically generated

Example

DELIMITER $$

CREATE PROCEDURE procedure\_name (in parameter)

BEGIN

SELECT …. ;

END$$

DELIMITER ;

PARAMETRIC PROCEDURES: Parametric statistics are based on assumptions about the distribution of population from which the sample was taken. Procedures with one input parameter can be used with aggregate functions too.

Example

USE emploees;

DROP PROCEDURE IF EXISTS emp\_salary;

DELIMITER $$

USE employees $$

CREATE PROCEDURE emp\_salary(IN p\_emp\_no INTEGER)

BEGIN

SELECT e.first\_name, e.last\_name, s.salary, s.from\_date, s.to\_date

FROM employees e

JOIN salaries s ON e.emp\_no = s.emp\_no

WHERE e.emp\_no = p\_emp\_no;

END $$

DELIMITER ;

CALL employees.emp\_salary(11300);

Example: Using Aggregate Functions

USE emploees;

DROP PROCEDURE IF EXISTS emp\_avg\_salary;

DELIMITER $$

USE employees $$

CREATE PROCEDURE emp\_avg\_salary(IN p\_emp\_no INTEGER)

BEGIN

SELECT e.first\_name, e.last\_name, AVG(s.salary)

FROM employees e

JOIN salaries s ON e.emp\_no = s.emp\_no

WHERE e.emp\_no = p\_emp\_no;

END $$

DELIMITER ;

CALL employees.emp\_avg\_salary(11300);

STORED PROCEDURES WITH AN OUTPUT PARAMETER: The out parameter will represent the variable containing the output value of the operation executed by the query of the stored procedure.

A picture containing bird, flower

Description automatically generated

**Note:** Everytime a procedure containing both an IN and an OUT parameter, we must use the SELECT-INTO structure.

Example

USE emploees;

DROP PROCEDURE IF EXISTS emp\_avg\_salary\_out;

DELIMITER $$

USE employees $$

CREATE PROCEDURE emp\_avg\_salary\_out(IN p\_emp\_no INTEGER, OUT p\_avg\_salary DECIMAL(10,2))

BEGIN

SELECT AVG(s.salary)

INTO p\_avg\_salary

FROM employees e

JOIN salaries s ON e.emp\_no = s.emp\_no

WHERE e.emp\_no = p\_emp\_no;

END $$

DELIMITER ;

CALL employees.emp\_avg\_salary\_out(11300, @p\_avg\_salary);

SELECT @p\_avg\_salary;

* Instead of CALL and SELECT, you may run the code by using the thunder sign next to the procedure.

SQL VARIABLES: When defining a program, such as a store procedure for instance, you can say you are using ‘parameters’ Parameters are a more abstract term, once the structure of the program has been solidified, then it will be applied to the database. The input value you insert is typically referred to as the ‘argument’, while the obtained output value is store in a ‘variable’. The name ‘variable’ is due to the fact that its content may vary depending on the input value used in the calculation.

A screenshot of a cell phone

Description automatically generated

To create a variable whose value equals the outcome of the calculation executed by the procedure we created, we need to write three queries:

1. Create a variable: SET
2. Extract a value that will be assigned to the newly created variable (call the procedure): CALL

Designate the place where the output value can be stored.

1. Ask the software to display the output of the procedure: SELECT

Example

SET @v\_avg\_salary = 0;

CALL employees.emp\_avg\_salary\_out(11300, @v\_avg\_salary);

SELECT @v\_avg\_salary;

**Note:**

* This is similar to the output seen when the thunder sign is used to run the procedure. That is because the process of implementing stored procedures in MySQL is unique.
* IN-OUT parameters exist, they are used when working with stored procedures and you need to override the content of a data point that has been used as an input with the output value obtained after running the calculation.

User-Defined Functions in MySQL Stored procedures are not the only type of stored routines available. Sometime, stored functions may be preferred. What’s the difference?

* Here there are no OUT parameters to define between the parentheses after the object’s name.
* All parameters are IN, and since this is well known, there is no need to explicitly indicate is with ‘IN’.
* Although there are no OUT parameters, there is a ‘return value’. This value can be of any data type.
* The return value is obtained after running the query contained in the body of the function.
* If the function returns an Error Code: 1418 use DETERMINISTIC, NO SQL, or READS SQL DATA.
* The error relates to a Binary law, use any of words provided above, after the CREATE FUNCTION line.

Example

USE employees;

DROP FUNCTION IF EXISTS f\_emp\_avg\_salary;

DELIMITER $$

USE employees $$

CREATE FUNCTION f\_emp\_avg\_salary (p\_emp\_no INTEGER) RETURNS DECIMAL(10,2)

DETERMINISTIC

BEGIN

DECLARE v\_avg\_salary DECIMAL(10,2);

SELECT AVG(s.salary)

INTO v\_avg\_salary

FROM employees e

JOIN salaries s ON e.emp\_no = s.emp\_no

WHERE e.emp\_no = p\_emp\_no;

RETURN v\_avg\_salary;

END $$

DELIMITER ;

SELECT f\_emp\_avg\_salary (11300);

Example

USE employees;

DROP FUNCTION IF EXISTS emp\_info;

DELIMITER $$

USE employees $$

CREATE FUNCTION emp\_info (p\_first\_name VARCHAR(200), p\_last\_name VARCHAR(200)) RETURNS DECIMAL(10,2)

DETERMINISTIC

BEGIN

DECLARE v\_max\_from\_date DATE;

DECLARE v\_salary DECIMAL(10,2);

SELECT MAX(s.from\_date)

INTO v\_max\_from\_date

FROM employees e

JOIN salaries s ON e.emp\_no = s.emp\_no

WHERE e.first\_name = p\_first\_name AND e.last\_name = p\_last\_name;

SELECT s.salary

INTO v\_salary

FROM employees e

JOIN salaries s ON e.emp\_no = s.emp\_no

WHERE e.first\_name = p\_first\_name AND e.last\_name = p\_last\_name AND s.from\_date = v\_max\_from\_date;

RETURN v\_salary;

END $$

DELIMITER ;

SELECT emp\_info('Aruna', 'Journel');

Technical Differences between Procedures and Functions

A screenshot of a cell phone

Description automatically generated

Conceptual Differences between Procedures and Functions

A screenshot of a cell phone

Description automatically generated

Summary:

* If you need to obtain more than one value as a result of a calculation, a Procedure it better off
* If you need just one value to be returned, then a Function is preferred.
* When considering other statements such as INSERT, UPDATE or DELETE,
* In these cases, the operation performed will apply changes to the data in the database.
* Thus, the INSERT, UPDATE or DELETE, statements should only be used in stored Procedures.
* Including a Procedure inside a SELECT statement is impossible.

**Note:** Functions may be included as of the columns inside a SELECT statement.

Example

SET @v\_emp\_no = 11300;

SELECT emp\_no, first\_name, last\_name,

F\_EMP\_AVG\_SALARY(@v\_emp\_no) AS avg\_salary

FROM employees

WHERE emp\_no = @v\_emp\_no;

The CASE statement: It is used within the SELECT statement when we want to return a specific value based on a condition.

Example

SELECT emp\_no, first\_name, last\_name,

CASE WHEN gender = 'M' THEN 'Male' ELSE 'Female'

END AS gender

FROM employees;

Example

SELECT emp\_no, first\_name, last\_name,

CASE gender WHEN 'M' THEN 'Male' ELSE 'Female'

END AS gender

FROM employees;

Example

SELECT e.emp\_no, e.first\_name, e.last\_name,

CASE WHEN dm.emp\_no IS NOT NULL THEN 'Manager' ELSE 'Employee'

END AS is\_manager

FROM employees e

LEFT JOIN dept\_manager dm ON dm.emp\_no = e.emp\_no

WHERE e.emp\_no > 109990;

Example

SELECT emp\_no, first\_name, last\_name,

IF (gender = 'M', 'Male', 'Female') AS gender

FROM employees;

IF vs. CASE

* With CASE, we can have multiple conditional expressions.
* Using IF, we can have just one conditional expression.
* See below

Example

SELECT dm.emp\_no, e.first\_name, e.last\_name,

MAX(s.salary) - MIN(s.salary) AS salary\_difference,

IF (MAX(s.salary) - MIN(s.salary) > 30000, 'Salary was raised by more than $30,000', 'Salary was

NOT raised by more than $30,000') AS salary\_raise

FROM dept\_manager dm

JOIN employees e ON e.emp\_no = dm.emp\_no

JOIN salaries s ON s.emp\_no = dm.emp\_no

GROUP BY s.emp\_no;

Example

SELECT dm.emp\_no, e.first\_name, e.last\_name, MAX(s.salary) - MIN(s.salary) AS salary\_difference,

CASE WHEN MAX(s.salary) - MIN(s.salary) > 30000

THEN 'Salary was raised by more than $30,000'

WHEN MAX(s.salary) - MIN(s.salary) BETWEEN 20000 AND 30000

THEN 'Salary was raised by more than $20,000 but less than $30,000'

ELSE 'Salary was raised by less than $20,000'

END AS salary\_increase

FROM dept\_manager dm

JOIN employees e ON e.emp\_no = dm.emp\_no

JOIN salaries s ON s.emp\_no = dm.emp\_no

GROUP BY s.emp\_no;