* **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

**SELECT DISTINCT Examples**

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

SELECT DISTINCT Country FROM Customers;

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

SELECT COUNT(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.

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

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Equal | [Try it](https://www.w3schools.com/sql/trysql.asp?filename=trysql_op_equal_to) |
| > | Greater than | [Try it](https://www.w3schools.com/sql/trysql.asp?filename=trysql_op_greater_than) |
| < | Less than | [Try it](https://www.w3schools.com/sql/trysql.asp?filename=trysql_op_less_than) |
| >= | Greater than or equal | [Try it](https://www.w3schools.com/sql/trysql.asp?filename=trysql_op_greater_than2) |
| <= | Less than or equal | [Try it](https://www.w3schools.com/sql/trysql.asp?filename=trysql_op_less_than2) |
| <> | Not equal. **Note:** In some versions of SQL this operator may be written as != | [Try it](https://www.w3schools.com/sql/trysql.asp?filename=trysql_op_not_equal_to) |
| BETWEEN | Between a certain range | [Try it](https://www.w3schools.com/sql/trysql.asp?filename=trysql_op_between) |
| LIKE | Search for a pattern | [Try it](https://www.w3schools.com/sql/trysql.asp?filename=trysql_op_like) |
| IN | To specify multiple possible values for a column | [Try it](https://www.w3schools.com/sql/trysql.asp?filename=trysql_op_in) |

## 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.

## 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';

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

## 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';

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

The following SQL statement selects all fields from "Customers" where country is "Germany" OR "Spain":

### **Example**

SELECT \* FROM Customers  
WHERE Country='Germany' OR Country='Spain';

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_where_or2)

## NOT Example

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

### **Example**

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

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

## 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.

## The SQL INSERT INTO Statement

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

## Insert Data Only in Specified Columns

It is also possible to only insert data in specific columns.

The following SQL statement will insert a new record, but only insert data in the "CustomerName", "City", and "Country" columns (CustomerID will be updated automatically):

### **Example**

INSERT INTO Customers (CustomerName, City, Country)  
VALUES ('Cardinal', 'Stavanger', '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.

**Note:** A NULL value is different from a zero value or a field that contains spaces. A field with a NULL value is one that has been left blank during record creation!

## 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;

## 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;

## The SQL UPDATE Statement

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

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

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;

## The SQL DELETE Statement

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

**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!

## 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 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 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 TOP, LIMIT and ROWNUM Examples

The following SQL statement selects the first three records from the "Customers" table: **SQL Server / MS Access Syntax:**

### **Example**

SELECT TOP 3 \* FROM Customers;

The following SQL statement shows the equivalent example using the LIMIT clause: **MySQL Syntax:**

### **Example**

SELECT \* FROM Customers  
LIMIT 3;

The following SQL statement shows the equivalent example using ROWNUM:

**Oracle Syntax:**

### **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() 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;

## The SQL 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.

## 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 one or more characters 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.

**Wildcard Characters in SQL Server**

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Description** | **Example** |
| % | Represents zero or more characters | bl% finds bl, black, blue, and blob |
| \_ | Represents a single character | h\_t finds hot, hat, and hit |
| [] | Represents any single character within the brackets | h[oa]t finds hot and hat, but not hit |
| ^ | Represents any character not in the brackets | h[^oa]t finds hit, but not hot and hat |
| - | Represents a range of characters | c[a-b]t finds cat and cbt |

All the wildcards can also be used in combinations!

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

|  |  |
| --- | --- |
| **LIKE Operator** | **Description** |
| WHERE CustomerName LIKE 'a%' | Finds any values that starts with "a" |
| WHERE CustomerName LIKE '%a' | Finds any values that ends with "a" |
| WHERE CustomerName LIKE '%or%' | Finds any values that have "or" in any position |
| WHERE CustomerName LIKE '\_r%' | Finds any values that have "r" in the second position |
| WHERE CustomerName LIKE 'a\_%\_%' | Finds any values that starts with "a" and are at least 3 characters in length |
| WHERE ContactName LIKE 'a%o' | Finds any values that starts with "a" and ends with "o" |

## 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 "ondon":

### **Example**

SELECT \* FROM Customers  
WHERE City LIKE '\_ondon';

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 Operator Examples

The following SQL statement selects all customers that are located in "Germany", "France" or "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 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.

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:

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;

## 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 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 INNER JOIN Keyword

The INNER JOIN keyword selects records that have matching values in both tables.



### **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!

## 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

The LEFT JOIN keyword 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.



## 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;

## SQL RIGHT JOIN Keyword

The RIGHT JOIN keyword 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.

**Note:** In some databases RIGHT JOIN is called RIGHT OUTER JOIN.



## 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;

**Note:** The RIGHT JOIN keyword returns all records from the right table (Employees), even if there are no matches in the left table (Orders).

## SQL FULL OUTER JOIN Keyword

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

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

**Tip:** FULL OUTER JOIN and FULL JOIN are the same.

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;

**Note:** The FULL OUTER JOIN keyword returns all matching records from both tables whether the other table matches or not. So, if there are rows in "Customers" that do not have matches in "Orders", or if there are rows in "Orders" that do not have matches in "Customers", those rows will be listed as well.

## SQL Self JOIN

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

## SQL Self JOIN Example

The following SQL statement matches customers that are from the same city:

### **Example**

SELECT A.CustomerName AS CustomerName1, B.CustomerName AS CustomerName2, A.City  
FROM Customers A, Customers B  
WHERE A.CustomerID <> B.CustomerID  
AND A.City = B.City  
ORDER BY A.City;

## The SQL UNION Operator

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

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

### **UNION Syntax**

SELECT column\_name(s) FROM table1  
UNION  
SELECT column\_name(s) FROM table2;

### **UNION ALL Syntax**

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

SELECT column\_name(s) FROM table1  
UNION ALL  
SELECT column\_name(s) FROM table2;

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

## Demo Database

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

Below is a selection from the "Customers" table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |

And a selection from the "Suppliers" table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **SupplierID** | **SupplierName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 1 | Exotic Liquid | Charlotte Cooper | 49 Gilbert St. | London | EC1 4SD | UK |
| 2 | New Orleans Cajun Delights | Shelley Burke | P.O. Box 78934 | New Orleans | 70117 | USA |
| 3 | Grandma Kelly's Homestead | Regina Murphy | 707 Oxford Rd. | Ann Arbor | 48104 | USA |

## SQL UNION Example

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

### **Example**

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

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

## SQL UNION ALL Example

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

### **Example**

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

## SQL UNION With WHERE

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

### **Example**

SELECT City, Country FROM Customers  
WHERE Country='Germany'  
UNION  
SELECT City, Country FROM Suppliers  
WHERE Country='Germany'  
ORDER BY City;

## SQL UNION ALL With WHERE

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

### **Example**

SELECT City, Country FROM Customers  
WHERE Country='Germany'  
UNION ALL  
SELECT City, Country FROM Suppliers  
WHERE Country='Germany'  
ORDER BY City;

## Another UNION Example

The following SQL statement lists all customers and suppliers:

### **Example**

SELECT 'Customer' As Type, ContactName, City, Country  
FROM Customers  
UNION  
SELECT 'Supplier', ContactName, City, Country  
FROM Suppliers;

Notice the "AS Type" above - it is an alias. [SQL Aliases](https://www.w3schools.com/sql/sql_alias.asp) are used to give a table or a column a temporary name. An alias only exists for the duration of the query. So, here we have created a temporary column named "Type", that list whether the contact person is a "Customer" or a "Supplier".

## The SQL GROUP BY Statement

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

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

### **Example**

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

## The SQL HAVING Clause

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

## The SQL EXISTS Operator

The EXISTS operator is used to test for the existence of any record in a subquery.

The EXISTS operator returns true if the subquery returns one or more records.

### **Example**

SELECT SupplierName  
FROM Suppliers  
WHERE EXISTS (SELECT ProductName FROM Products WHERE Products.SupplierID = Suppliers.supplierID AND Price = 22);

## The SQL ANY and ALL Operators

The ANY and ALL operators are used with a WHERE or HAVING clause.

The ANY operator returns true if any of the subquery values meet the condition.

The ALL operator returns true if all of the subquery values meet the condition.

## SQL ANY Examples

The ANY operator returns TRUE if any of the subquery values meet the condition.

The following SQL statement returns TRUE and lists the product names if it finds ANY records in the OrderDetails table that quantity = 10:

### **Example**

SELECT ProductName  
FROM Products  
WHERE ProductID = ANY (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);

The following SQL statement returns TRUE and lists the product names if it finds ANY records in the OrderDetails table that quantity > 99:

### **Example**

SELECT ProductName  
FROM Products  
WHERE ProductID = ANY (SELECT ProductID FROM OrderDetails WHERE Quantity > 99);

## SQL ALL Example

The ALL operator returns TRUE if all of the subquery values meet the condition.

The following SQL statement returns TRUE and lists the product names if ALL the records in the OrderDetails table has quantity = 10:

### **Example**

SELECT ProductName  
FROM Products  
WHERE ProductID = ALL (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);

### **SELECT INTO Syntax**

Copy all columns into a new table:

## The SQL INSERT INTO SELECT Statement

The INSERT INTO SELECT statement copies data from one table and inserts it into another table.

* INSERT INTO SELECT requires that data types in source and target tables match
* The existing records in the target table are unaffected

### **Example**

INSERT INTO Customers (CustomerName, City, Country)  
SELECT SupplierName, City, Country FROM Suppliers;

## The SQL CASE Statement

The CASE statement goes through conditions and returns a value when the first condition is met (like an IF-THEN-ELSE statement). So, once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in the ELSE clause.

### **Example**

SELECT OrderID, Quantity,  
CASE  
    WHEN Quantity > 30 THEN "The quantity is greater than 30"  
    WHEN Quantity = 30 THEN "The quantity is 30"  
    ELSE "The quantity is under 30"  
END AS QuantityText  
FROM OrderDetails;

## What is a Stored Procedure?

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.

You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

The following SQL statement creates a stored procedure named "SelectAllCustomers" that selects all records from the "Customers" table:

### **Example**

CREATE PROCEDURE SelectAllCustomers  
AS  
SELECT \* FROM Customers  
GO;

### **Example**

EXEC SelectAllCustomers;

The CREATE DATABASE statement is used to create a new SQL database.

### **Syntax**

CREATE DATABASE databasename;

**Tip:** Make sure you have admin privilege before creating any database. Once a database is created, you can check it in the list of databases with the following SQL command: SHOW DATABASES;

## The SQL DROP DATABASE Statement

The DROP DATABASE statement is used to drop an existing SQL database.

### **Syntax**

DROP DATABASE databasename;

**Note:** Be careful before dropping a database. Deleting a database will result in loss of complete information stored in the database!

## BACKUP DATABASE Example

The following SQL statement creates a full back up of the existing database "testDB" to the D disk:

### **Example**

BACKUP DATABASE testDB  
TO DISK = 'D:\backups\testDB.bak';

**Tip:** Always back up the database to a different drive than the actual database. Then, if you get a disk crash, you will not lose your backup file along with the database.

## BACKUP WITH DIFFERENTIAL Example

The following SQL statement creates a differential back up of the database "testDB":

### **Example**

BACKUP DATABASE testDB  
TO DISK = 'D:\backups\testDB.bak'  
WITH DIFFERENTIAL;

**Tip:** A differential back up reduces the back up time (since only the changes are backed up).

## SQL CREATE TABLE Example

The following example creates a table called "Persons" that contains five columns: PersonID, LastName, FirstName, Address, and City:

### **Example**

CREATE TABLE Persons (  
    PersonID int,  
    LastName varchar(255),  
    FirstName varchar(255),  
    Address varchar(255),  
    City varchar(255)  
);

## The SQL DROP TABLE Statement

The DROP TABLE statement is used to drop an existing table in a database.

### **Syntax**

DROP TABLE table\_name;

**Note:** Be careful before dropping a table. Deleting a table will result in loss of complete information stored in the table!

## SQL DROP TABLE Example

The following SQL statement drops the existing table "Shippers":

### **Example**

DROP TABLE Shippers;

## SQL TRUNCATE TABLE

The TRUNCATE TABLE statement is used to delete the data inside a table, but not the table itself.

### **Syntax**

TRUNCATE TABLE table\_name;

## SQL ALTER TABLE Statement

The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.

The ALTER TABLE statement is also used to add and drop various constraints on an existing table.

## ALTER TABLE - ADD Column

To add a column in a table, use the following syntax:

ALTER TABLE table\_name  
ADD column\_name datatype;

The following SQL adds an "Email" column to the "Customers" table:

### **Example**

ALTER TABLE Customers  
ADD Email varchar(255);

## ALTER TABLE - DROP COLUMN

To delete a column in a table, use the following syntax (notice that some database systems don't allow deleting a column):

ALTER TABLE table\_name  
DROP COLUMN column\_name;

The following SQL deletes the "Email" column from the "Customers" table:

### **Example**

ALTER TABLE Customers  
DROP COLUMN Email;

## ALTER TABLE - ALTER/MODIFY COLUMN

To change the data type of a column in a table, use the following syntax:

**My SQL / Oracle (prior version 10G):**

ALTER TABLE table\_name  
MODIFY COLUMN column\_name datatype;

**Oracle 10G and later:**

ALTER TABLE table\_name  
MODIFY column\_name datatype;

DROP COLUMN Example

Next, we want to delete the column named "DateOfBirth" in the "Persons" table.

We use the following SQL statement:

ALTER TABLE Persons  
DROP COLUMN DateOfBirth;

## SQL Constraints

SQL constraints are used to specify rules for the data in a table.

Constraints are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted.

Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

The following constraints are commonly used in SQL:

* [**NOT NULL**](https://www.w3schools.com/sql/sql_notnull.asp) - Ensures that a column cannot have a NULL value
* [**UNIQUE**](https://www.w3schools.com/sql/sql_unique.asp) - Ensures that all values in a column are different
* [**PRIMARY KEY**](https://www.w3schools.com/sql/sql_primarykey.asp) - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
* [**FOREIGN KEY**](https://www.w3schools.com/sql/sql_foreignkey.asp) - Uniquely identifies a row/record in another table
* [**CHECK**](https://www.w3schools.com/sql/sql_check.asp) - Ensures that all values in a column satisfies a specific condition
* [**DEFAULT**](https://www.w3schools.com/sql/sql_default.asp) - Sets a default value for a column when no value is specified
* [**INDEX**](https://www.w3schools.com/sql/sql_create_index.asp) - Used to create and retrieve data from the database very quickly

## SQL NOT NULL Constraint

By default, a column can hold NULL values.

The NOT NULL constraint enforces a column to NOT accept NULL values.

This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.

## SQL NOT NULL on CREATE TABLE

The following SQL ensures that the "ID", "LastName", and "FirstName" columns will NOT accept NULL values when the "Persons" table is created:

### **Example**

CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255) NOT NULL,  
    Age int  
);

## SQL NOT NULL on ALTER TABLE

To create a NOT NULL constraint on the "Age" column when the "Persons" table is already created, use the following SQL:

ALTER TABLE Persons  
MODIFY Age int NOT NULL;

# **SQL UNIQUE Constraint**

[❮ Previous](https://www.w3schools.com/sql/sql_notnull.asp)[Next ❯](https://www.w3schools.com/sql/sql_primarykey.asp)

## SQL UNIQUE Constraint

The UNIQUE constraint ensures that all values in a column are different.

Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns.

A PRIMARY KEY constraint automatically has a UNIQUE constraint.

However, you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

## SQL UNIQUE Constraint on CREATE TABLE

The following SQL creates a UNIQUE constraint on the "ID" column when the "Persons" table is created:

**SQL Server / Oracle / MS Access:**

CREATE TABLE Persons (  
    ID int NOT NULL UNIQUE,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int  
);

**MySQL:**

CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    UNIQUE (ID)  
);

**MySQL / SQL Server / Oracle / MS Access:**

CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    CONSTRAINT UC\_Person UNIQUE (ID,LastName)  
);

SQL UNIQUE Constraint on ALTER TABLE

To create a UNIQUE constraint on the "ID" column when the table is already created, use the following SQL:

**MySQL / SQL Server / Oracle / MS Access:**

ALTER TABLE Persons  
ADD UNIQUE (ID);

To name a UNIQUE constraint, and to define a UNIQUE constraint on multiple columns, use the following SQL syntax:

**MySQL / SQL Server / Oracle / MS Access:**

ALTER TABLE Persons  
ADD CONSTRAINT UC\_Person UNIQUE (ID,LastName);

DROP a UNIQUE Constraint

To drop a UNIQUE constraint, use the following SQL:

**MySQL:**

ALTER TABLE Persons  
DROP INDEX UC\_Person;

**SQL Server / Oracle / MS Access:**

ALTER TABLE Persons  
DROP CONSTRAINT UC\_Person;

SQL PRIMARY KEY Constraint

The PRIMARY KEY constraint uniquely identifies each record in a table.

Primary keys must contain UNIQUE values, and cannot contain NULL values.

A table can have only ONE primary key; and in the table, this primary key can consist of single or multiple columns (fields).

SQL PRIMARY KEY on CREATE TABLE

The following SQL creates a PRIMARY KEY on the "ID" column when the "Persons" table is created:

**MySQL:**

CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    PRIMARY KEY (ID)  
);

**SQL Server / Oracle / MS Access:**

CREATE TABLE Persons (  
    ID int NOT NULL PRIMARY KEY,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int  
);

To allow naming of a PRIMARY KEY constraint, and for defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

**MySQL / SQL Server / Oracle / MS Access:**

CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    CONSTRAINT PK\_Person PRIMARY KEY (ID,LastName)  
);

**Note:** In the example above there is only ONE PRIMARY KEY (PK\_Person). However, the VALUE of the primary key is made up of TWO COLUMNS (ID + LastName).

SQL PRIMARY KEY on ALTER TABLE

To create a PRIMARY KEY constraint on the "ID" column when the table is already created, use the following SQL:

**MySQL / SQL Server / Oracle / MS Access:**

ALTER TABLE Persons  
ADD PRIMARY KEY (ID);

To allow naming of a PRIMARY KEY constraint, and for defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

**MySQL / SQL Server / Oracle / MS Access:**

ALTER TABLE Persons  
ADD CONSTRAINT PK\_Person PRIMARY KEY (ID,LastName);

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

DROP a PRIMARY KEY Constraint

To drop a PRIMARY KEY constraint, use the following SQL:

**MySQL:**

ALTER TABLE Persons  
DROP PRIMARY KEY;

**SQL Server / Oracle / MS Access:**

ALTER TABLE Persons  
DROP CONSTRAINT PK\_Person;

SQL FOREIGN KEY Constraint

A FOREIGN KEY is a key used to link two tables together.

A FOREIGN KEY is a field (or collection of fields) in one table that refers to the PRIMARY KEY in another table.

The table containing the foreign key is called the child table, and the table containing the candidate key is called the referenced or parent table.

Look at the following two tables:

"Persons" table:

|  |  |  |  |
| --- | --- | --- | --- |
| **PersonID** | **LastName** | **FirstName** | **Age** |
| 1 | Hansen | Ola | 30 |
| 2 | Svendson | Tove | 23 |
| 3 | Pettersen | Kari | 20 |

"Orders" table:

|  |  |  |
| --- | --- | --- |
| **OrderID** | **OrderNumber** | **PersonID** |
| 1 | 77895 | 3 |
| 2 | 44678 | 3 |
| 3 | 22456 | 2 |
| 4 | 24562 | 1 |

Notice that the "PersonID" column in the "Orders" table points to the "PersonID" column in the "Persons" table.

The "PersonID" column in the "Persons" table is the PRIMARY KEY in the "Persons" table.

The "PersonID" column in the "Orders" table is a FOREIGN KEY in the "Orders" table.

The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.

The FOREIGN KEY constraint also prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the table it points to.

SQL FOREIGN KEY on CREATE TABLE

The following SQL creates a FOREIGN KEY on the "PersonID" column when the "Orders" table is created:

**MySQL:**

CREATE TABLE Orders (  
    OrderID int NOT NULL,  
    OrderNumber int NOT NULL,  
    PersonID int,  
    PRIMARY KEY (OrderID),  
    FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)  
);

**SQL Server / Oracle / MS Access:**

CREATE TABLE Orders (  
    OrderID int NOT NULL PRIMARY KEY,  
    OrderNumber int NOT NULL,  
    PersonID int FOREIGN KEY REFERENCES Persons(PersonID)  
);

SQL CHECK Constraint

The CHECK constraint is used to limit the value range that can be placed in a column.

If you define a CHECK constraint on a single column it allows only certain values for this column.

If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

SQL CHECK on CREATE TABLE

The following SQL creates a CHECK constraint on the "Age" column when the "Persons" table is created. The CHECK constraint ensures that you can not have any person below 18 years:

**MySQL:**

CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    CHECK (Age>=18)  
);

SQL DEFAULT Constraint

The DEFAULT constraint is used to provide a default value for a column.

The default value will be added to all new records IF no other value is specified.

SQL DEFAULT on CREATE TABLE

The following SQL sets a DEFAULT value for the "City" column when the "Persons" table is created:

**My SQL / SQL Server / Oracle / MS Access:**

CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    City varchar(255) DEFAULT 'Sandnes'  
);

The DEFAULT constraint can also be used to insert system values, by using functions like GETDATE():

CREATE TABLE Orders (  
    ID int NOT NULL,  
    OrderNumber int NOT NULL,  
    OrderDate date DEFAULT GETDATE()  
);

SQL DEFAULT on ALTER TABLE

To create a DEFAULT constraint on the "City" column when the table is already created, use the following SQL:

**MySQL:**

ALTER TABLE Persons  
ALTER City SET DEFAULT 'Sandnes';

**SQL Server:**

ALTER TABLE Persons  
ADD CONSTRAINT df\_City  
DEFAULT 'Sandnes' FOR City;

## SQL CREATE INDEX Statement

The CREATE INDEX statement is used to create indexes in tables.

Indexes are used to retrieve data from the database very fast. The users cannot see the indexes, they are just used to speed up searches/queries.

**Note:** Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So, only create indexes on columns that will be frequently searched against.

### **CREATE INDEX Syntax**

Creates an index on a table. Duplicate values are allowed:

CREATE INDEX index\_name  
ON table\_name (column1, column2, ...);

### **CREATE UNIQUE INDEX Syntax**

Creates a unique index on a table. Duplicate values are not allowed:

CREATE UNIQUE INDEX index\_name  
ON table\_name (column1, column2, ...);

**Note:** The syntax for creating indexes varies among different databases. Therefore: Check the syntax for creating indexes in your database.

## CREATE INDEX Example

The SQL statement below creates an index named "idx\_lastname" on the "LastName" column in the "Persons" table:

CREATE INDEX idx\_lastname  
ON Persons (LastName);

If you want to create an index on a combination of columns, you can list the column names within the parentheses, separated by commas:

CREATE INDEX idx\_pname  
ON Persons (LastName, FirstName);

## DROP INDEX Statement

The DROP INDEX statement is used to delete an index in a table.

**MS Access:**

DROP INDEX index\_name ON table\_name;

**SQL Server:**

DROP INDEX table\_name.index\_name;

**DB2/Oracle:**

DROP INDEX index\_name;

**MySQL:**

ALTER TABLE table\_nameDROP INDEX index\_name;

## SQL CREATE VIEW Statement

In SQL, a view is a virtual table based on the result-set of an SQL statement.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.

### **CREATE VIEW Syntax**

CREATE VIEW view\_name AS  
SELECT column1, column2, ...  
FROM table\_name  
WHERE condition;

**Note:** A view always shows up-to-date data! The database engine recreates the data, using the view's SQL statement, every time a user queries a view.

## SQL CREATE VIEW Examples

The following SQL creates a view that shows all customers from Brazil:

### **Example**

CREATE VIEW [Brazil Customers] AS  
SELECT CustomerName, ContactName  
FROM Customers  
WHERE Country = "Brazil";

We can query the view above as follows:

### **Example**

SELECT \* FROM [Brazil Customers];