SQL

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

## 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.

## RDBMS

RDBMS stands for Relational Database Management System.

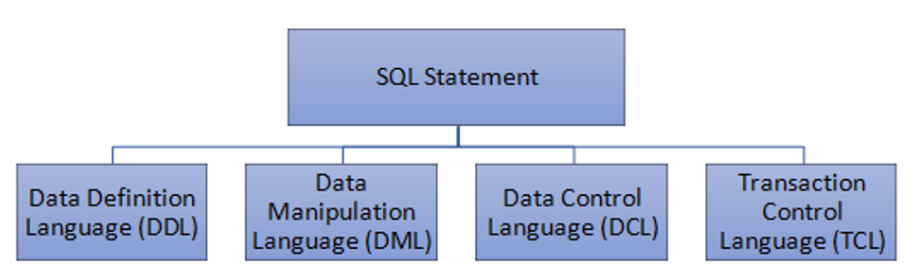
RDBMS is the basis for SQL, and for all modern database systems such as MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.

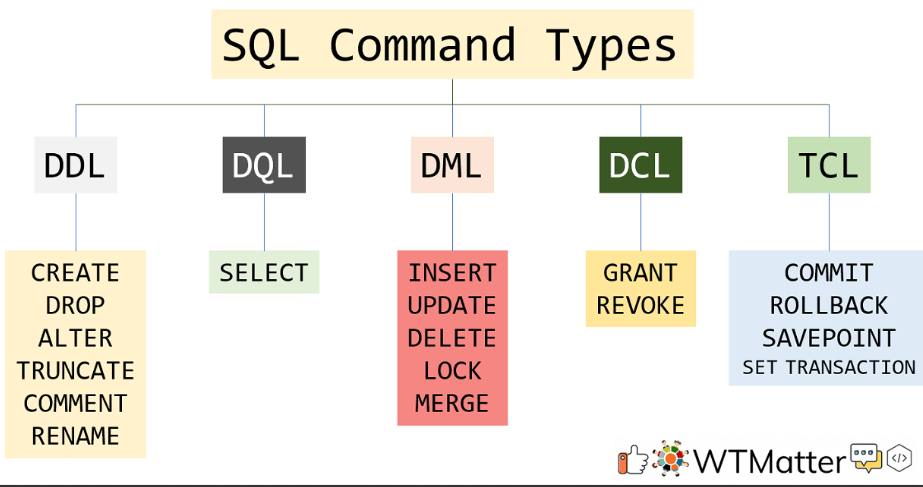
The data in RDBMS is stored in database objects called tables. A table is a collection of related data entries and it consists of columns and rows.

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.





## ORDER BY

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

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

## The SQL INSERT INTO Statement

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

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

## 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.

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

## The SQL UPDATE Statement

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

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

## UPDATE Multiple Records

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

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

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

## The SQL DELETE Statement

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

DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste';

## The SQL MIN() and MAX() Functions

The MIN() function returns the smallest value of the selected column.

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 criterion.

SELECT COUNT(ProductID)  
FROM Products;

## AVG() Example

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

SELECT AVG(Price)  
FROM Products;

## SUM() Example

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

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 often used in conjunction with the LIKE operator:

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

|  |  |
| --- | --- |
| **LIKE Operator** | **Description** |
| WHERE CustomerName LIKE 'a%' | Finds any values that start with "a" |
| WHERE CustomerName LIKE '%a' | Finds any values that end 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 start with "a" and are at least 2 characters in length |
| WHERE CustomerName LIKE 'a\_\_%' | Finds any values that start with "a" and are at least 3 characters in length |
| WHERE ContactName LIKE 'a%o' | Finds any values that start with "a" and ends with "o" |

## SQL LIKE Examples

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

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

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

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

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

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

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

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:

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":

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

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

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 [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 MS Access |  |
| **Symbol** | **Description** |
| \* | Represents zero or more characters |
| ? | Represents a single character |
| [] | Represents any single character within the brackets |
| ! | Represents any character not in the brackets |
| - | Represents any single character within the specified range |
| # | Represents any single numeric character |

|  |  |  |
| --- | --- | --- |
| 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 any single character within the specified range | c[a-b]t finds cat and cbt |

## 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.

SELECT \* FROM Customers  
WHERE Country IN ('Germany', 'France', 'UK');

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:

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.

### **BETWEEN Syntax**

SELECT column\_name(s)  
FROM table\_name  
WHERE column\_name BETWEEN value1 AND value2;

## 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 that query.

An alias is created with the AS keyword.

### **Alias Column Syntax**

SELECT column\_name AS alias\_name  
FROM table\_name;

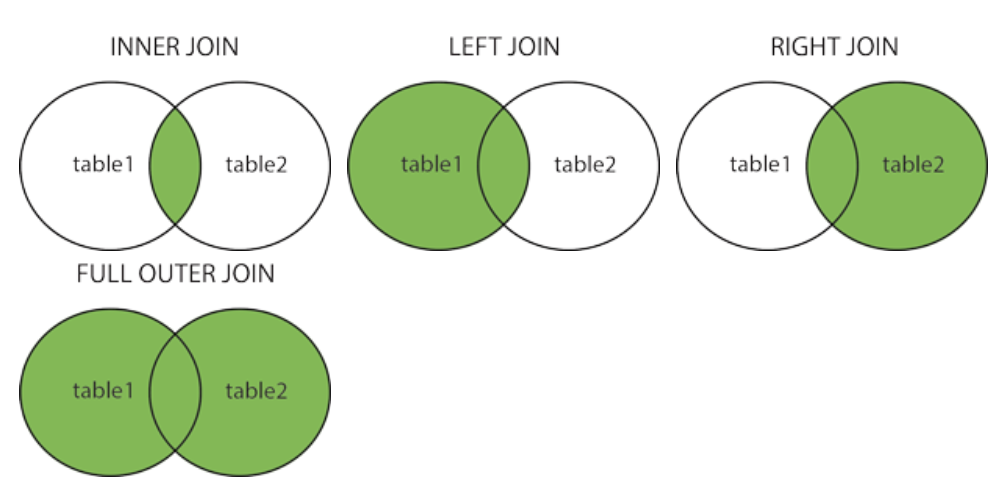
## SQL JOIN

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

## 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: Returns all records from the left table, and the matched records from the right table
* RIGHT (OUTER) JOIN: Returns all records from the right table, and the matched records from the left table
* FULL (OUTER) JOIN: Returns 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 in both tables.

### **INNER JOIN Syntax**

SELECT column\_name(s)  
FROM table1  
INNER JOIN table2ON table1.column\_name = table2.column\_name;

## SQL LEFT JOIN Keyword

The LEFT JOIN keyword returns all records from the left table (table1), and the matching records from the right table (table2). The result is 0 records from the right side, if there is no match.

### **LEFT JOIN Syntax**

SELECT column\_name(s)  
FROM table1  
LEFT JOIN table2ON table1.column\_name = table2.column\_name;

## SQL RIGHT JOIN Keyword

The RIGHT JOIN keyword returns all records from the right table (table2), and the matching records from the left table (table1). The result is 0 records from the left side, if there is no match.

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

### **RIGHT JOIN Syntax**

SELECT column\_name(s)  
FROM table1  
RIGHT JOIN table2ON table1.column\_name = table2.column\_name;

## SQL FULL OUTER JOIN Keyword

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

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

### **FULL OUTER JOIN Syntax**

SELECT column\_name(s)  
FROM table1  
FULL OUTER JOIN table2ON table1.column\_name = table2.column\_nameWHERE condition;

## 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;

## The SQL UNION Operator

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

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

### **UNION Syntax**

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

## 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.

### **GROUP BY Syntax**

SELECT column\_name(s)  
FROM table\_name  
WHERE condition  
GROUP BY column\_name(s)ORDER BY column\_name(s);

## The SQL HAVING Clause

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

### **HAVING Syntax**

SELECT column\_name(s)  
FROM table\_name  
WHERE condition  
GROUP BY column\_name(s)HAVING conditionORDER BY column\_name(s);

## 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.

### **EXISTS Syntax**

SELECT column\_name(s)  
FROM table\_name  
WHERE EXISTS  
(SELECT column\_name FROM table\_name WHERE condition);

## The SQL ANY Operator

The ANY operator:

* returns a boolean value as a result
* returns TRUE if ANY of the subquery values meet the condition

ANY means that the condition will be true if the operation is true for any of the values in the range.

### **ANY Syntax**

SELECT column\_name(s)  
FROM table\_name  
WHERE column\_name operator ANY  
  (SELECT column\_name  FROM table\_name  WHERE condition);

## The SQL ALL Operator

The ALL operator:

* returns a boolean value as a result
* returns TRUE if ALL of the subquery values meet the condition
* is used with SELECT, WHERE and HAVING statements

ALL means that the condition will be true only if the operation is true for all values in the range.

### **ALL Syntax With SELECT**

SELECT ALL column\_name(s)  
FROM table\_name  
WHERE condition;

## The SQL SELECT INTO Statement

The SELECT INTO statement copies data from one table into a new table.

### **SELECT INTO Syntax**

Copy all columns into a new table:

SELECT \*  
INTO newtable [IN externaldb]  
FROM oldtableWHERE condition;

Copy only some columns into a new table:

SELECT *column1*, *column2*, *column3*, ...  
INTO *newtable* [IN *externaldb*]  
FROM *oldtable*WHERE *condition;*

## The SQL INSERT INTO SELECT Statement

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

The INSERT INTO SELECT statement requires that the data types in source and target tables match.

**Note:** The existing records in the target table are unaffected.

### **INSERT INTO SELECT Syntax**

Copy all columns from one table to another table:

INSERT INTO table2  
SELECT \* FROM table1WHERE condition;

Copy only some columns from one table into another table:

INSERT INTO table2 (column1, column2, column3, ...)  
SELECT column1, column2, column3, ...  
FROM table1  
WHERE condition;

The SQL CASE Expression

The CASE expression 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.

If there is no ELSE part and no conditions are true, it returns NULL.

CASE Syntax

CASE  
    WHEN *condition1* THEN *result1*  
    WHEN *condition2* THEN *result2*  
    WHEN *conditionN* THEN *resultN*  
    ELSE *result*  
END;

## SQL CASE Examples

**The following SQL goes through conditions and returns a value when the first condition is met:**

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;

**The following SQL will order the customers by City. However, if City is NULL, then order by Country:**

SELECT CustomerName, City, Country  
FROM Customers  
ORDER BY  
(CASE  
    WHEN City IS NULL THEN Country  
    ELSE City  
END);

## 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.

### **Stored Procedure Syntax**

CREATE PROCEDURE procedure\_name  
AS  
sql\_statement  
GO;

## Stored Procedure Example

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;

Execute the stored procedure above as follows:

### **Example**

EXEC SelectAllCustomers;

## Single Line Comments

Single line comments start with --.

Any text between -- and the end of the line will be ignored (will not be executed).

The following example uses a single-line comment as an explanation:

### **Example**

--Select all:  
SELECT \* FROM Customers;

## SQL Arithmetic Operators

|  |  |
| --- | --- |
| SQL Arithmetic Operators |  |
| **Operator** | **Description** |
| + | Add |
| - | Subtract |
| \* | Multiply |
| / | Divide |
| % | Modulo |

## SQL Bitwise Operators

|  |  |
| --- | --- |
| **Operator** | **Description** |
| & | Bitwise AND |
| | | Bitwise OR |
| ^ | Bitwise exclusive OR |

## SQL Comparison Operators

|  |  |
| --- | --- |
| **Operator** | **Description** |
| = | Equal to |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal to |
| <= | Less than or equal to |
| <> | Not equal to |

## SQL Compound Operators

|  |  |
| --- | --- |
| **Operator** | **Description** |
| += | Add equals |
| -= | Subtract equals |
| \*= | Multiply equals |
| /= | Divide equals |
| %= | Modulo equals |
| &= | Bitwise AND equals |
| ^-= | Bitwise exclusive equals |
| |\*= | Bitwise OR equals |

## SQL Logical Operators

|  |  |
| --- | --- |
| **Operator** | **Description** |
| ALL | TRUE if all of the subquery values meet the condition |
| AND | TRUE if all the conditions separated by AND is TRUE |
| ANY | TRUE if any of the subquery values meet the condition |
| BETWEEN | TRUE if the operand is within the range of comparisons |
| EXISTS | TRUE if the subquery returns one or more records |
| IN | TRUE if the operand is equal to one of a list of expressions |
| LIKE | TRUE if the operand matches a pattern |
| NOT | Displays a record if the condition(s) is NOT TRUE |
| OR | TRUE if any of the conditions separated by OR is TRUE |
| SOME | TRUE if any of the subquery values meet the condition |

## The SQL BACKUP DATABASE Statement

The BACKUP DATABASE statement is used in SQL Server to create a full back up of an existing SQL database.

### **Syntax**

BACKUP DATABASE databasename  
TO DISK = 'filepath';

## 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) - Prevents actions that would destroy links between tables
* [CHECK](https://www.w3schools.com/sql/sql_check.asp) - Ensures that the values in a column satisfies a specific condition
* [DEFAULT](https://www.w3schools.com/sql/sql_default.asp) - Sets a default value for a column if no value is specified
* [CREATE INDEX](https://www.w3schools.com/sql/sql_create_index.asp) - Used to create and retrieve data from the database very quickly

## SQL CREATE INDEX Statement

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

Indexes are used to retrieve data from the database more quickly than otherwise. 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);

## SQL Date Data Types

**MySQL** comes with the following data types for storing a date or a date/time value in the database:

* DATE - format YYYY-MM-DD
* DATETIME - format: YYYY-MM-DD HH:MI:SS
* TIMESTAMP - format: YYYY-MM-DD HH:MI:SS
* YEAR - format YYYY or YY

**SQL Server** comes with the following data types for storing a date or a date/time value in the database:

* DATE - format YYYY-MM-DD
* DATETIME - format: YYYY-MM-DD HH:MI:SS
* SMALLDATETIME - format: YYYY-MM-DD HH:MI:SS
* TIMESTAMP - format: a unique number

**Note:** The date types are chosen for a column when you create a new table in your database!

## SQL CREATE VIEW Statement

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

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

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

A view is created with the CREATE VIEW statement.

### **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 view, every time a user queries it.

## 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];

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_view2)

The following SQL creates a view that selects every product in the "Products" table with a price higher than the average price:

### **Example**

CREATE VIEW [Products Above Average Price] AS  
SELECT ProductName, Price  
FROM Products  
WHERE Price > (SELECT AVG(Price) FROM Products);

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_view3)

We can query the view above as follows:

### **Example**

SELECT \* FROM [Products Above Average Price];

## SQL Updating a View

A view can be updated with the CREATE OR REPLACE VIEW statement.

### **SQL CREATE OR REPLACE VIEW Syntax**

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

The following SQL adds the "City" column to the "Brazil Customers" view:

### **Example**

CREATE OR REPLACE VIEW [Brazil Customers] AS  
SELECT CustomerName, ContactName, City  
FROM Customers  
WHERE Country = 'Brazil';

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_view5)

## SQL Dropping a View

A view is deleted with the DROP VIEW statement.

### **SQL DROP VIEW Syntax**

DROP VIEW view\_name;

The following SQL drops the "Brazil Customers" view:

### **Example**

DROP VIEW [Brazil Customers];

## SQL Injection

SQL injection is a code injection technique that might destroy your database.

SQL injection is one of the most common web hacking techniques.

SQL injection is the placement of malicious code in SQL statements, via web page input.

## SQL in Web Pages

SQL injection usually occurs when you ask a user for input, like their username/userid, and instead of a name/id, the user gives you an SQL statement that you will **unknowingly** run on your database.

Look at the following example which creates a SELECT statement by adding a variable (txtUserId) to a select string. The variable is fetched from user input (getRequestString):

### **Example**

txtUserId = getRequestString("UserId");  
txtSQL = "SELECT \* FROM Users WHERE UserId = " + txtUserId;

The rest of this chapter describes the potential dangers of using user input in SQL statements.

## SQL Injection Based on 1=1 is Always True

Look at the example above again. The original purpose of the code was to create an SQL statement to select a user, with a given user id.

If there is nothing to prevent a user from entering "wrong" input, the user can enter some "smart" input like this:

UserId: 

Then, the SQL statement will look like this:

SELECT \* FROM Users WHERE UserId = 105 OR 1=1;

The SQL above is valid and will return ALL rows from the "Users" table, since **OR 1=1** is always TRUE.

Does the example above look dangerous? What if the "Users" table contains names and passwords?

The SQL statement above is much the same as this:

SELECT UserId, Name, Password FROM Users WHERE UserId = 105 or 1=1;

A hacker might get access to all the user names and passwords in a database, by simply inserting 105 OR 1=1 into the input field.

## SQL Injection Based on ""="" is Always True

Here is an example of a user login on a web site:

Username:  


Password:  


### **Example**

uName = getRequestString("username");  
uPass = getRequestString("userpassword");  
  
sql = 'SELECT \* FROM Users WHERE Name ="' + uName + '" AND Pass ="' + uPass + '"'

### **Result**

SELECT \* FROM Users WHERE Name ="John Doe" AND Pass ="myPass"

A hacker might get access to user names and passwords in a database by simply inserting " OR ""=" into the user name or password text box:

User Name:  


Password:  


The code at the server will create a valid SQL statement like this:

### **Result**

SELECT \* FROM Users WHERE Name ="" or ""="" AND Pass ="" or ""=""

The SQL above is valid and will return all rows from the "Users" table, since **OR ""=""** is always TRUE.

## SQL Injection Based on Batched SQL Statements

Most databases support batched SQL statement.

A batch of SQL statements is a group of two or more SQL statements, separated by semicolons.

The SQL statement below will return all rows from the "Users" table, then delete the "Suppliers" table.

### **Example**

SELECT \* FROM Users; DROP TABLE Suppliers

Look at the following example:

### **Example**

txtUserId = getRequestString("UserId");  
txtSQL = "SELECT \* FROM Users WHERE UserId = " + txtUserId;

And the following input:

User id: 

The valid SQL statement would look like this:

### **Result**

SELECT \* FROM Users WHERE UserId = 105; DROP TABLE Suppliers;

## Use SQL Parameters for Protection

To protect a web site from SQL injection, you can use SQL parameters.

SQL parameters are values that are added to an SQL query at execution time, in a controlled manner.

### **ASP.NET Razor Example**

txtUserId = getRequestString("UserId");  
txtSQL = "SELECT \* FROM Users WHERE UserId = @0";  
db.Execute(txtSQL,txtUserId);

Note that parameters are represented in the SQL statement by a @ marker.

The SQL engine checks each parameter to ensure that it is correct for its column and are treated literally, and not as part of the SQL to be executed.

### **Another Example**

txtNam = getRequestString("CustomerName");  
txtAdd = getRequestString("Address");  
txtCit = getRequestString("City");  
txtSQL = "INSERT INTO Customers (CustomerName,Address,City) Values(@0,@1,@2)";  
db.Execute(txtSQL,txtNam,txtAdd,txtCit);

## Examples

The following examples shows how to build parameterized queries in some common web languages.

SELECT STATEMENT IN ASP.NET:

txtUserId = getRequestString("UserId");  
sql = "SELECT \* FROM Customers WHERE CustomerId = @0";  
command = new SqlCommand(sql);  
command.Parameters.AddWithValue("@0",txtUserId);  
command.ExecuteReader();

INSERT INTO STATEMENT IN ASP.NET:

txtNam = getRequestString("CustomerName");  
txtAdd = getRequestString("Address");  
txtCit = getRequestString("City");  
txtSQL = "INSERT INTO Customers (CustomerName,Address,City) Values(@0,@1,@2)";  
command = new SqlCommand(txtSQL);  
command.Parameters.AddWithValue("@0",txtNam);  
command.Parameters.AddWithValue("@1",txtAdd);  
command.Parameters.AddWithValue("@2",txtCit);  
command.ExecuteNonQuery();

INSERT INTO STATEMENT IN PHP:

$stmt = $dbh->prepare("INSERT INTO Customers (CustomerName,Address,City)  
VALUES (:nam, :add, :cit)");  
$stmt->bindParam(':nam', $txtNam);  
$stmt->bindParam(':add', $txtAdd);  
$stmt->bindParam(':cit', $txtCit);  
$stmt->execute();

**Summary**: in this tutorial, you will learn about SQL window functions that solve complex query challenges in easy ways.

## Introduction to SQL Window Functions

The [aggregate functions](https://www.sqltutorial.org/sql-aggregate-functions/) perform calculations across a set of rows and return a single output row.

The following query uses the [SUM()](https://www.sqltutorial.org/sql-aggregate-functions/sql-sum/) aggregate function to calculate the total salary of all employees in the company:

SELECT

SUM(salary) sum\_salary

FROM

employees;

Code language: SQL (Structured Query Language) (sql)

Here is the output:

https://www.sqltutorial.org/wp-content/uploads/2018/09/sql-window-functions-sum-aggregate-function.png

As shown clearly in the output, all rows from the  employees table are grouped into a single row.

Similar to an aggregate function, a window function calculates on a set of rows. However, a window function does not cause rows to become grouped into a single output row.

The following query uses the SUM() as a window function. It returns the sum salary of all employees along with the salary of each individual employee:

SELECT

first\_name,

last\_name,

salary,

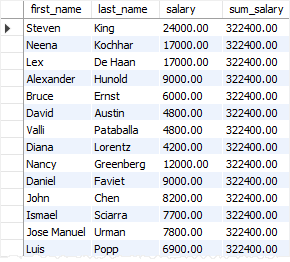
SUM(salary) OVER() sum\_salary

FROM

employees;

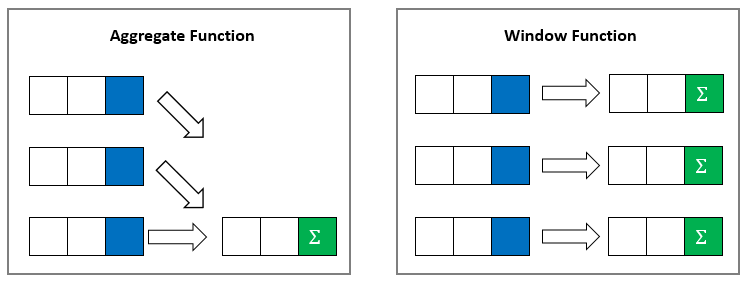
Code language: SQL (Structured Query Language) (sql)

Here is the partial output:



In this example, the OVER() clause signals that the SUM() function is used as a window function.

The following picture illustrates the main difference between aggregate functions and window functions:



## SQL window function syntax

The syntax of the window functions is as follows:

window\_function\_name ( expression ) OVER (

partition\_clause

order\_clause

frame\_clause

)

Code language: SQL (Structured Query Language) (sql)

window\_function\_name

The name of the supported window function such as [ROW\_NUMBER()](https://www.sqltutorial.org/sql-window-functions/sql-row_number/), [RANK()](https://www.sqltutorial.org/sql-window-functions/sql-rank/), and [SUM()](https://www.sqltutorial.org/sql-aggregate-functions/sql-sum/).

expression

The target expression or column on which the window function operates.

OVER clause

The OVER clause defines window partitions to form the groups of rows specifies the orders of rows in a partition. The OVER clause consists of three clauses: partition, order, and frame clauses.

The partition clause divides the rows into partitions to which the window function applies. It has the following syntax:

PARTITION BY expr1, expr2, ...

Code language: SQL (Structured Query Language) (sql)

If the PARTITION BY clause is not specified, then the whole result set is treated as a single partition.

The order clause specifies the orders of rows in a partition on which the window function operates:

ORDER BY

expression [ASC | DESC] [NULL {FIRST| LAST}]

,...

Code language: SQL (Structured Query Language) (sql)

A frame is the subset of the current partition. To define the frame, you use one of the following syntaxes:

{ RANGE | ROWS } frame\_start

{ RANGE | ROWS } BETWEEN frame\_start AND frame\_end

Code language: SQL (Structured Query Language) (sql)

where frame\_start is one of the following options:

N PRECEDING

UNBOUNDED PRECEDING

CURRENT ROW

Code language: SQL (Structured Query Language) (sql)

and frame\_end is one of the following options:

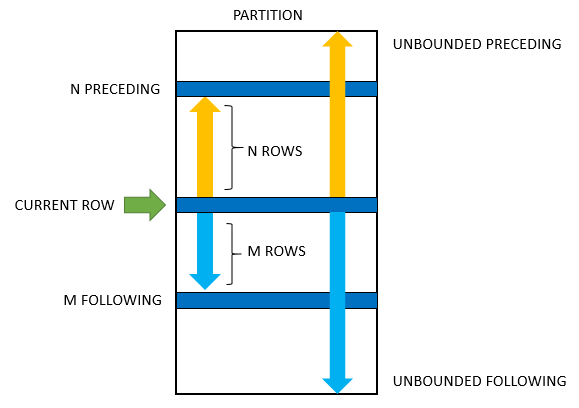
CURRENT ROW

UNBOUNDED FOLLOWING

N FOLLOWING

Code language: SQL (Structured Query Language) (sql)

The following picture illustrates a frame and its options:



* UNBOUNDED PRECEDING: the frame starts at the first row of the partition.
* N PRECEDING: the frame starts at Nth rows before the current row.
* CURRENT ROW: means the current row that is being evaluated.
* UNBOUNDED FOLLOWING: the frame ends at the final row in the partition.
* N FOLLOWING: the frame ends at the Nh row after the current row.

The ROWS or RANGE specifies the type of relationship between the current row and frame rows.

* ROWS: the offsets of the current row and frame rows are row numbers.
* RANGE: the offset of the current row and frame rows are row values.

## SQL window function types

The window functions are divided into three types value window functions, aggregation window functions, and ranking window functions:

Value window functions

* [FIRST\_VALUE()](https://www.sqltutorial.org/sql-window-functions/sql-first_value/)
* [LAG()](https://www.sqltutorial.org/sql-window-functions/sql-lag/)
* [LAST\_VALUE()](https://www.sqltutorial.org/sql-window-functions/sql-last_value/)
* [LEAD()](https://www.sqltutorial.org/sql-window-functions/sql-lead/)

Ranking window functions

* [CUME\_DIST()](https://www.sqltutorial.org/sql-window-functions/sql-cume_dist/)
* [DENSE\_RANK()](https://www.sqltutorial.org/sql-window-functions/sql-dense_rank/)
* [NTILE()](https://www.sqltutorial.org/sql-window-functions/sql-ntile/)
* [PERCENT\_RANK()](https://www.sqltutorial.org/sql-window-functions/sql-percent_rank/)
* [RANK()](https://www.sqltutorial.org/sql-window-functions/sql-rank/)
* [ROW\_NUMBER()](https://www.sqltutorial.org/sql-window-functions/sql-row_number/)

Aggregate window functions

* AVG()
* COUNT()
* MAX()
* MIN()
* SUM()

# TRIGGER

A TRIGGER, also known as a trigger, is a kind of database SQL programming script. The TRIGGERs are procedures that will be executed according to our instructions when operations are carried out on the information in the database. These operations can be update (UPDATE), insertion (INSERT) and deletion (DELETE).

CREATE TRIGGER trigger\_name

ON origin\_table

AFTER INSERT, DELETE

AS

BEGIN

SET NOCOUNT ON;

INSERT INTO target\_table(

row,

)

SELECT

i.origin\_row,

GETDATE(),

'INS'

FROM

inserted i

UNION ALL

SELECT

d.origin\_row,

GETDATE(),

'DEL'

FROM

deleted d;

END

The common table expression

The common table expression (CTE) is a powerful construct in SQL that helps simplify a query. CTEs work as virtual tables (with records and columns), created during the execution of a query, used by the query, and eliminated after query execution. CTEs often act as a bridge to transform the data in source tables to the format expected by the query.

A **common table expression**, or CTE, is a temporary named result set created from a simple SELECT statement that can be used in a subsequent SELECT statement. Each SQL CTE is like a **named query**, whose result is stored in a virtual table (a CTE) to be referenced later in the main query.

The best way to learn common table expressions is through practice. I recommend LearnSQL.com's interactive [Recursive Queries](https://learnsql.com/course/common-table-expressions) course. It contains over 100 exercises that teach CTEs starting with the basics and progressing to advanced topics like recursive common table expressions.

WITH my\_cte AS (

  SELECT a,b,c

  FROM T1

)

SELECT a,c

FROM my\_cte

WHERE ....

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **branch** | **date** | **seller** | **item** | **quantity** | **unit\_price** |
| Paris-1 | 12/7/2021 | Charles | Headphones A2 | 1 | 80 |
| London-1 | 12/6/2021 | John | Cell Phone X2 | 2 | 120 |
| London-2 | 12/7/2021 | Mary | Headphones A1 | 1 | 60 |
| Paris-1 | 12/7/2021 | Charles | Battery Charger | 1 | 50 |
| London-2 | 12/7/2021 | Mary | Cell Phone B2 | 2 | 90 |
| London-1 | 12/7/2021 | John | Headphones A0 | 5 | 75 |
| London-1 | 12/7/2021 | Sean | Cell Phone X1 | 2 | 100 |

In the first example, we obtain a report with the same records from the **sales** table but add an extra column with the price of the most expensive item sold in the same branch that day. To obtain the price of the most expensive item, we use a common table expression like this:

|  |
| --- |
| WITH highest AS ( |
| SELECT |
| branch, |
| date, |
| MAX(unit\_price) AS highest\_price |
| FROM sales |
| GROUP BY branch, date |
| ) |
| SELECT |
| sales.\*, |
| h.highest\_price |
| FROM sales |
| JOIN highest h |
| ON sales.branch = h.branch |
| AND sales.date = h.date |