# SQL Queries

## Insert Into

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

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

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

## Update

UPDATE *table\_name*  
SET *column1*=*value1*,*column2*=*value2*,...  
WHERE *some\_column*=*some\_value*;

**Notice the WHERE clause in the SQL UPDATE statement!**  
The WHERE clause specifies which record or records that should be updated. If you omit the WHERE clause, all records will be updated!

Change the value of the "City" column of a record in the "Customers" table:

UPDATE Customers  
SET City='Hamburg'  
WHERE CustomerID=1;

UPDATE Multiple Columns

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

UPDATE Multiple Records

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

## Delete

DELETE statement is used to delete records in a table.

DELETE FROM *table\_name*  
WHERE *some\_column*=*some\_value*;

**Notice the WHERE clause in the SQL DELETE statement!**  
The WHERE clause specifies which record or records that should be deleted. If you omit the WHERE clause, all records will be deleted!

DELETE FROM Customers  
WHERE CustomerName='Alfreds Futterkiste' AND ContactName='Maria Anders';

**Delete All Data**

DELETE FROM table\_name;  
  
or  
  
DELETE \* FROM table\_name;

## SELECT DISTINCT

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

SELECT DISTINCT Country FROM Customers;

SELECT COUNT(DISTINCT Country) FROM Customers;

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

## WHERE

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

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

SELECT \* FROM Customers  
WHERE CustomerID=1;

## AND OR NOT

### AND Syntax

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

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

### OR Syntax

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

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

### NOT Syntax

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

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

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

## ORDER BY Syntax

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

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

SELECT \* FROM Customers  
ORDER BY Country;

SELECT \* FROM Customers  
ORDER BY Country, CustomerName;

SELECT \* FROM Customers  
ORDER BY Country DESC;

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

## IS NULL Operator

The following SQL statement uses the IS NULL operator to list all persons that have no address:

SELECT LastName, FirstName, Address FROM Persons  
WHERE Address IS NULL;

## IS NOT NULL Operator

SELECT LastName, FirstName, Address FROM Persons  
WHERE Address IS NOT NULL;

## MIN() and MAX() Functions

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

SELECT MIN(Price) AS SmallestPrice  
FROM Products;

The MAX() function returns the largest value of the selected column.

SELECT MAX(Price) AS LargestPrice  
FROM Products;

## COUNT(), AVG() and SUM() Functions

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

SELECT COUNT(ProductID)  
FROM Products;

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

SELECT AVG(Price)  
FROM Products;

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

SELECT SUM(Quantity)  
FROM OrderDetails;

## 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

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

|  |
| --- |
| **City** |
| Aachen |
| Albuquerque |
| Anchorage |
| Ann Arbor |

## GROUP BY

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

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

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

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

## HAVING Syntax

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

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

SELECT COUNT(CustomerID), Country  
FROM Customers  
GROUP BY Country  
HAVING COUNT(CustomerID) > 5;

SELECT COUNT(CustomerID), Country  
FROM Customers  
GROUP BY Country  
HAVING COUNT(CustomerID) > 5  
ORDER BY COUNT(CustomerID) DESC;

**SQL Injection**

**SQL in Web Pages**

In the previous chapters, you have learned to retrieve (and update) database data, using SQL.

When SQL is used to display data on a web page, it is common to let web users input their own search values.

Since SQL statements are text only, it is easy, with a little piece of computer code, to dynamically change SQL statements to provide the user with selected data:

UserId:   


Server Result

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

## Select Top

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

SELECT TOP 2 \* FROM Customers;

SELECT TOP 50 PERCENT \* FROM Customers;

## LIKE Operator

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

**SQL LIKE Syntax**

SELECT column\_name(s)  
FROM table\_name  
WHERE column\_name LIKE pattern;

SELECT \* FROM Customers  
WHERE City LIKE 'Chenn%';

SELECT \* FROM Customers  
WHERE Country LIKE '%angolr%';

Using the NOT keyword allows you to select records that do NOT match the pattern.

The following SQL statement selects all customers with Country NOT containing the pattern "land":

SELECT \* FROM Customers  
WHERE Country NOT LIKE '%land%';

## SQL Wildcards

A wildcard character can be used to substitute for any other character(s) in a string.

**Wildcard Description**

% A substitute for zero or more characters

\_ A substitute for a single character

[charlist] Sets and ranges of characters to match

[^charlist] Matches only a character NOT specified within the brackets

or

[!charlist]

SELECT \* FROM Customers  
WHERE City LIKE 'ber%';

SELECT \* FROM Customers  
WHERE City LIKE '%es%';

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

SELECT \* FROM Customers  
WHERE City LIKE 'L\_n\_on';

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

SELECT \* FROM Customers  
WHERE City LIKE '[bsp]%';

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

SELECT \* FROM Customers  
WHERE City LIKE '[a-c]%';

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

SELECT \* FROM Customers  
WHERE City LIKE '[!bsp]%';

SELECT \* FROM Customers  
WHERE City NOT LIKE '[bsp]%';

## IN Operator

The IN operator allows you to specify multiple values in a WHERE clause.

SELECT column\_name(s)  
FROM table\_name  
WHERE column\_name IN (value1,value2,...);

SELECT \* FROM Customers  
WHERE City IN ('Paris','London');

## BETWEEN Operator

The BETWEEN operator selects values within a range. The values can be numbers, text, or dates.

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

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

SELECT \* FROM Products  
WHERE Price NOT BETWEEN 10 AND 20;

SELECT \* FROM Products  
WHERE (Price BETWEEN 10 AND 20)  
AND NOT CategoryID IN (1,2,3);

SELECT \* FROM Products  
WHERE ProductName BETWEEN 'C' AND 'M';

SELECT \* FROM Products  
WHERE ProductName NOT BETWEEN 'C' AND 'M';

## SQL Aliases

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

Basically aliases are created to make column names more readable.

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

SELECT column\_name(s)  
FROM table\_nameAS alias\_name;

SELECT CustomerName AS Customer, ContactName AS [Contact Person]  
FROM Customers;

|  |  |
| --- | --- |
| **Customer** | **Contact Person** |
| Alfreds Futterkiste | Maria Anders |
| Ana Trujillo Emparedados y helados | Ana Trujillo |
| Antonio Moreno Taquería | Antonio Moreno |
|  |  |

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

SELECT CustomerName, Address+', '+City+', '+PostalCode+', '+Country AS Address  
FROM Customers;

|  |  |
| --- | --- |
| **CustomerName** | **Address** |
| Alfreds Futterkiste | Obere Str. 57, Berlin, 12209, Germany |
| Ana Trujillo Emparedados y helados | Avda. de la Constitución 2222, México D.F., 05021, Mexico |

*Alias Example for Tables:*

SELECT o.OrderID, o.OrderDate, c.CustomerName  
FROM Customers AS c, Orders AS o  
WHERE c.CustomerName="Around the Horn" ANDc.CustomerID=o.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 joins are used to combine rows from two or more tables.

**SQL JOIN**

An SQL JOIN clause is used to combine rows from two or more tables, based on a common field between them.

The most common type of join is: **SQL INNER JOIN (simple join)**. An SQL INNER JOIN returns all rows from multiple tables where the join condition is met.

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

**Different SQL JOINs**

Before we continue with examples, we will list the types of the different SQL JOINs you can use:

* **INNER JOIN**: Returns all rows when there is at least one match in BOTH tables
* **LEFT JOIN**: Return all rows from the left table, and the matched rows from the right table
* **RIGHT JOIN**: Return all rows from the right table, and the matched rows from the left table
* **FULL JOIN**: Return all rows when there is a match in ONE of the tables

**SQL INNER JOIN Keyword**

The INNER JOIN keyword selects all rows from both tables as long as there is a match between the columns in both tables.

**SQL INNER JOIN Syntax**

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

or:

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

**PS!** INNER JOIN is the same as JOIN.



SELECT Customers.CustomerName, Orders.OrderID  
FROM Customers  
INNER JOIN Orders  
ON Customers.CustomerID=Orders.CustomerID  
ORDER BY Customers.CustomerName;

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

## SQL LEFT JOIN Keyword

The LEFT JOIN keyword returns all rows from the left table (table1), with the matching rows in the right table (table2). The result is NULL in the right side when there is no match.

### SQL LEFT JOIN Syntax

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

or:

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

**PS!** In some databases LEFT JOIN is called LEFT OUTER JOIN.



SELECT Customers.CustomerName, Orders.OrderID  
FROM Customers  
LEFT JOIN Orders  
ON Customers.CustomerID=Orders.CustomerID  
ORDER BY Customers.CustomerName;

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

## SQL RIGHT JOIN Keyword

The RIGHT JOIN keyword returns all rows from the right table (table2), with the matching rows in the left table (table1). The result is NULL in the left side when there is no match.

### SQL RIGHT JOIN Syntax

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

or:

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

**PS!** In some databases RIGHT JOIN is called RIGHT OUTER JOIN.



SELECT Orders.OrderID, Employees.FirstName  
FROM Orders  
RIGHT JOIN Employees  
ON Orders.EmployeeID=Employees.EmployeeID  
ORDER BY Orders.OrderID;

**Note:** The RIGHT JOIN keyword returns all the rows 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 returns all rows from the left table (table1) and from the right table (table2).

The FULL OUTER JOIN keyword combines the result of both LEFT and RIGHT joins.

### SQL FULL OUTER JOIN Syntax

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



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 the rows from the left table (Customers), and all the rows from the right table (Orders). 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 UNION Operator**

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

Notice that each SELECT statement within the UNION must have the same number of columns. The columns must also have similar data types. Also, the columns in each SELECT statement must be in the same order.

**SQL UNION Syntax**

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

**Note:** The UNION operator selects only distinct values by default. To allow duplicate values, use the ALL keyword with UNION.

**SQL UNION ALL Syntax**

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

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

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

**Note:** UNION cannot be used to list ALL cities from the two tables. If several customers and suppliers share the same city, each city will only be listed once. UNION selects only distinct values. Use UNION ALL to also select duplicate values!

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

**UNION ALL With WHERE**

The following SQL statement uses UNION ALL to select **all** (duplicate values also) **German** cities from the "Customers" and "Suppliers" tables:

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

**SQL SELECT INTO Statement**

The SELECT INTO statement selects data from one table and inserts it into a new table.

We can copy all columns into the new table:

SELECT \*  
INTO newtable [IN externaldb]  
FROM table1;

Or we can copy only the columns we want into the new table:

SELECT column\_name(s)  
INTO newtable [IN externaldb]  
FROM table1;

The new table will be created with the column-names and types as defined in the SELECT statement. You can apply new names using the AS clause.

**SQL SELECT INTO Examples**

Create a backup copy of Customers:

SELECT \*  
INTO CustomersBackup2013  
FROM Customers;

Use the IN clause to copy the table into another database:

SELECT \*  
INTO CustomersBackup2013 IN 'Backup.mdb'  
FROM Customers;

Copy only a few columns into the new table:

SELECT CustomerName, ContactName  
INTO CustomersBackup2013  
FROM Customers;

Copy only the German customers into the new table:

SELECT \*  
INTO CustomersBackup2013  
FROM Customers  
WHERE Country='Germany';

Copy data from more than one table into the new table:

SELECT Customers.CustomerName, Orders.OrderID  
INTO CustomersOrderBackup2013  
FROM Customers  
LEFT JOIN Orders  
ON Customers.CustomerID=Orders.CustomerID;

**Tip:** The SELECT INTO statement can also be used to create a new, empty table using the schema of another. Just add a WHERE clause that causes the query to return no data:

SELECT \*  
INTO newtable  
FROM table1  
WHERE 1=0;

**SQL INSERT INTO SELECT Statement**

The INSERT INTO SELECT statement selects data from one table and inserts it into an existing table. Any existing rows in the target table are unaffected.

We can copy all columns from one table to another, existing table:

INSERT INTO table2  
SELECT \* FROM table1;

Or we can copy only the columns we want to into another, existing table:

INSERT INTO table2  
(column\_name(s))  
SELECT column\_name(s)  
FROM table1;

Copy only a few columns from "Suppliers" into "Customers":

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

Copy only the German suppliers into "Customers":

INSERT INTO Customers (CustomerName, Country)  
SELECT SupplierName, Country FROM Suppliers  
WHERE Country='Germany';

**CREATE DATABASE Statement**

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

CREATE DATABASE databasename;

CREATE DATABASE testDB;

**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;

**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!

DROP DATABASE testDB;

**SQL CREATE TABLE Statement**

The CREATE TABLE statement is used to create a new table in a database.

CREATE TABLE table\_name(  
    column1 datatype,  
    column2 datatype,  
    column3 datatype,  
   ....  
);

The column parameters specify the names of the columns of the table.

The datatype parameter specifies the type of data the column can hold (e.g. varchar, integer, date, etc.).

**Tip:** For an overview of the available data types, go to our complete [Data Types Reference](https://www.w3schools.com/sql/sql_datatypes.asp).

**SQL CREATE TABLE Example**

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

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

**Create Table Using Another Table**

A copy of an existing table can be created using a combination of the CREATE TABLE statement and the SELECT statement.

The new table gets the same column definitions. All columns or specific columns can be selected.

If you create a new table using an existing table, the new table will be filled with the existing values from the old table.

CREATE TABLE new\_table\_name AS  
    SELECT column1, column2,...  
    FROM existing\_table\_name  
    WHERE ....;

**SQL DROP TABLE Statement**

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

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!

DROP TABLE Shippers;

**SQL TRUNCATE TABLE**

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

TRUNCATE TABLE table\_name;

**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*;

ALTER TABLE Persons  
ADD DateOfBirth date;

**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*;

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

ALTER TABLE Persons  
DROP COLUMN DateOfBirth;

**ALTER TABLE - ALTER/MODIFY COLUMN**

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

SQL Server / MS Access:

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

My SQL / Oracle (prior version 10G):

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

we want to change the data type of the column named "DateOfBirth" in the "Persons" table.

ALTER TABLE Persons  
ALTER COLUMN DateOfBirth year;

**Create Constraints**

Constraints can be specified when the table is created with the CREATE TABLE statement, or after the table is created with the ALTER TABLE statement.

CREATE TABLE table\_name(  
    column1 datatype constraint,  
    column2 datatype constraint,  
    column3 datatype constraint,  
    ....  
);

**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) - Use to create and retrieve data from the database very quickly

**NOT NULL Constraint**

By default, a column can hold NULL values. The NOT NULL constraint enforces a column to NOT accept NULL values.

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

The following SQL ensures that the "ID", "LastName", and "FirstName" columns will NOT accept NULL values:

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

**Tip:** If the table has already been created, you can add a NOT NULL constraint to a column with the [ALTER TABLE](https://www.w3schools.com/sql/sql_alter.asp) statement.

**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)  
);

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:**

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 database table.

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

A table can have only one primary key, which may consist of single or multiple 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;