# DML

## Distinct

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

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

SELECT COUNT(DISTINCT Country) FROM Customers;

## Where

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

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

## Order By

SELECT \* FROM Customers  
ORDER BY Country;

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

## Insert

INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)

VALUES ('Cardinal','Tom B. Erichsen','Skagen 21','Stavanger','4006','Norway');

## IS NULL

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

## UPDATE

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

## DELETE

DELETE FROM Customers  
WHERE CustomerName='Alfreds Futterkiste';

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

DELETE FROM table\_name;

## TOP

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

SELECT TOP 5 \* FROM Products order by Price desc SELECT TOP 50 PERCENT \* FROM Customers;

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

## Min Max

SELECT MAX(Price) AS LargestPrice  
FROM Products;

Count AVG SUM

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.

SELECT COUNT(ProductID)  
FROM Products;

SELECT AVG(Price)  
FROM Products;

SELECT SUM(Quantity)  
FROM OrderDetails;

## Like

|  |  |
| --- | --- |
| **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" |

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

## Wildcards

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]%';

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

## IN

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

he 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);

SELECT Distinct Country as UniqCountry FROM Customers

WHERE Country in (SELECT Country FROM Suppliers);

## BETWEEN

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

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

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

SELECT \* FROM Products  
WHERE ProductName BETWEEN 'Carnarvon Tigers' AND 'Mozzarella di Giovanni'  
ORDER BY ProductName;

The following SQL statement selects all orders with an OrderDate BETWEEN '04-July-1996' and '09-July-1996'

SELECT \* FROM Orders  
WHERE OrderDate BETWEEN #07/04/1996# AND #07/09/1996#;

## Aliases

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

SELECT CustomerName AS Customer, ContactName AS [Contact Person]  
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

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

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)

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

SELECT Orders.OrderID, Orders.OrderDate, Customers.CustomerName  
FROM Customers, Orders  
WHERE Customers.CustomerName="Around the Horn" AND Customers.CustomerID=Orders.CustomerID;

## JOIN

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

Here are the different types of the JOINs in SQL:

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

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);

## LEFT JOIN

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

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

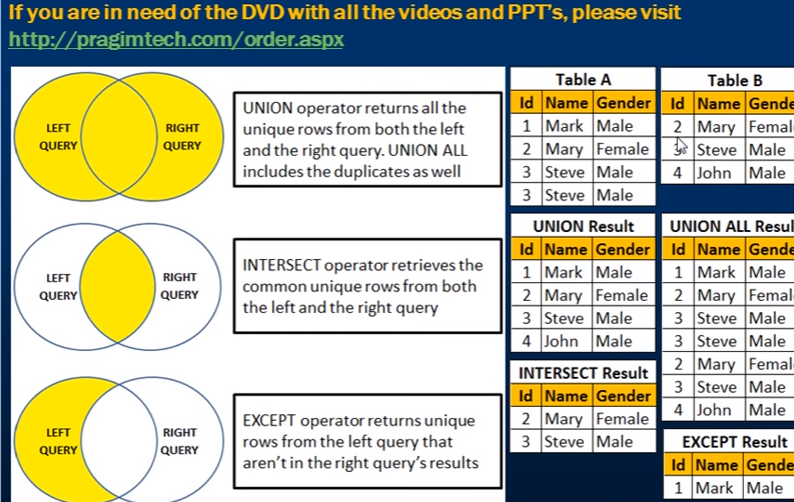
## FULL OUTTER JOIN

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

## SELF JOIN

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;

## UNION



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

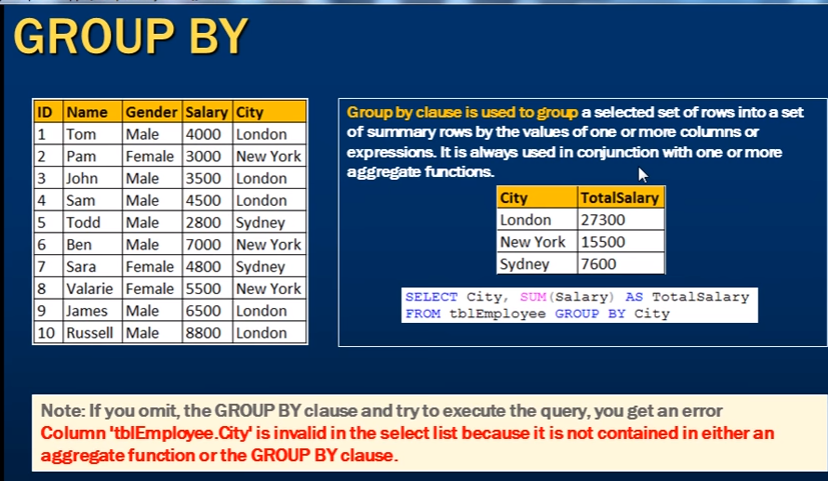
SELECT City FROM Customers  
UNION  
SELECT City FROM Suppliers  
ORDER BY City;

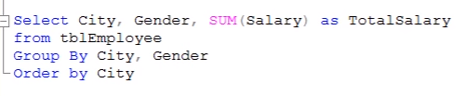
SELECT City FROM Customers  
UNION ALL  
SELECT City FROM Suppliers  
ORDER BY City;

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

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

## GROUP BY





The following SQL statement lists the number of customers in each country

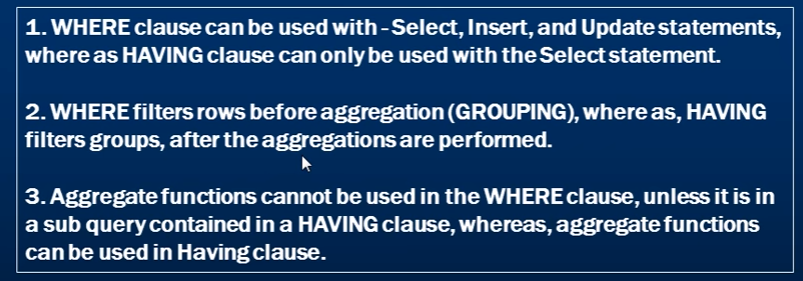
SELECT COUNT(CustomerID), Country  
FROM Customers  
GROUP BY Country;

The following SQL statement lists the number of customers in each country, sorted high to low

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

SELECT Shippers.ShipperName, COUNT(Orders.OrderID) AS NumberOfOrders FROM Orders  
LEFT JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID  
GROUP BY ShipperName;

## HAVING



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;

SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders  
FROM (Orders  
INNER JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID)  
GROUP BY LastName  
HAVING COUNT(Orders.OrderID) > 10;

## EXIST

SELECT SupplierName  
FROM Suppliers  
WHERE EXISTS (SELECT ProductName FROM Products WHERE SupplierId = Suppliers.supplierId AND Price < 20);

## ANY

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

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

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

SELECT INTO

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

The following SQL statement creates a backup copy of Customers:

SELECT \* INTO CustomersBackup2017  
FROM Customers;

The following SQL statement uses the IN clause to copy the table into a new table in another database:

SELECT \* INTO CustomersBackup2017 IN 'Backup.mdb'  
FROM Customers;

The following SQL statement copies only a few columns into a new table:

SELECT CustomerName, ContactName INTO CustomersBackup2017  
FROM Customers;

The following SQL statement copies only the German customers into a new table:

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

SELECT INTO 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 *oldtable*  
WHERE 1 = 0;

## INSERT INTO SELECT

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

# SQL Database

## DROP DB

The following SQL statement drops the existing database "testDB":

DROP DATABASE testDB;

## CREATE TABLE

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

## DROP TABLE

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

DROP TABLE table\_name;

## ALTER TABLE

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.

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*;

Now we want to add a column named "DateOfBirth" in the "Persons" table.

We use the following SQL statement:

ALTER TABLE Persons  
ADD DateOfBirth date;

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

We use the following SQL statement:

ALTER TABLE Persons  
ALTER COLUMN DateOfBirth year;

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;

## CONSTRAINTS

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

QL 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

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

### UNIQUE

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.

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

ALTER TABLE Persons  
ADD UNIQUE (ID);

ALTER TABLE Persons  
DROP CONSTRAINT UC\_Person;

### PRIMARY KEY

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

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)  
);

o 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 drop a PRIMARY KEY constraint, use the following SQL

ALTER TABLE Persons  
DROP CONSTRAINT PK\_Person;

### FOREIGNKEY

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.

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

To create a FOREIGN KEY constraint on the "PersonID" column when the "Orders" table is already created, use the following SQL:

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

ALTER TABLE Orders  
ADD FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);

To drop a FOREIGN KEY constraint, use the following SQL

ALTER TABLE Orders  
DROP CONSTRAINT FK\_PersonOrder;

### CHECK

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.

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

CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    City varchar(255),  
    CONSTRAINT CHK\_Person CHECK (Age>=18 AND City='Sandnes')  
);

To create a CHECK constraint on the "Age" column when the table is already created, use the following SQL

ALTER TABLE Persons  
ADD CHECK (Age>=18);

ALTER TABLE Persons  
DROP CONSTRAINT CHK\_PersonAge;

### DEFAULT

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.

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

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

ALTER TABLE Persons  
ALTER COLUMN City SET DEFAULT 'Sandnes';

ALTER TABLE Persons  
ALTER COLUMN City DROP DEFAULT;

### INDEX

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 index\_name ON table\_name;

# AUTO INCREMENT

Auto-increment allows a unique number to be generated automatically when a new record is inserted into a table.

Often this is the primary key field that we would like to be created automatically every time a new record is inserted.

The following SQL statement defines the "ID" column to be an auto-increment primary key field in the "Persons" table:

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

The following SQL statement defines the "ID" column to be an auto-increment primary key field in the "Persons" table:

CREATE TABLE Persons (  
    ID int IDENTITY(1,1) PRIMARY KEY,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int  
);

The following SQL statement defines the "ID" column to be an auto-increment primary key field in the "Persons" table:

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

# VIEWS

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.

The view "Current Product List" lists all active products (products that are not discontinued) from the "Products" table. The view is created with the following SQL:

CREATE VIEW [Current Product List] AS  
SELECT ProductID, ProductName  
FROM Products  
WHERE Discontinued = No;

Another view in the Northwind sample database selects every product in the "Products" table with a unit price higher than the average unit price:

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

Another view in the Northwind database calculates the total sale for each category in 1997. Note that this view selects its data from another view called "Product Sales for 1997":

CREATE VIEW [Category Sales For 1997] AS  
SELECT DISTINCT CategoryName, Sum(ProductSales) AS CategorySales  
FROM [Product Sales for 1997]  
GROUP BY CategoryName;

We can query the view above as follows:

SELECT \* FROM [Category Sales For 1997];

You can update a view by using the following syntax:

**SQL CREATE OR REPLACE VIEW Syntax**

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

Now we want to add the "Category" column to the "Current Product List" view. We will update the view with the following SQL:

CREATE OR REPLACE VIEW [Current Product List] AS  
SELECT ProductID, ProductName, Category  
FROM Products  
WHERE Discontinued = No;

**SQL DROP VIEW Syntax**

DROP VIEW view\_name;

# 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.

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;

# DATES

**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!

# OPERATORS

## **SQL Logical Operators**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| ALL | TRUE if all of the subquery values meet the condition | [Try it](https://www.w3schools.com/SQL/trysql.asp?filename=trysql_op_all&ss=-1) |
| AND | TRUE if all the conditions separated by AND is TRUE | [Try it](https://www.w3schools.com/SQL/trysql.asp?filename=trysql_op_and) |
| ANY | TRUE if any of the subquery values meet the condition | [Try it](https://www.w3schools.com/SQL/trysql.asp?filename=trysql_op_any&ss=-1) |
| BETWEEN | TRUE if the operand is within the range of comparisons | [Try it](https://www.w3schools.com/SQL/trysql.asp?filename=trysql_op_between) |
| EXISTS | TRUE if the subquery returns one or more records | [Try it](https://www.w3schools.com/SQL/trysql.asp?filename=trysql_op_exists) |
| IN | TRUE if the operand is equal to one of a list of expressions | [Try it](https://www.w3schools.com/SQL/trysql.asp?filename=trysql_op_in) |
| LIKE | TRUE if the operand matches a pattern | [Try it](https://www.w3schools.com/SQL/trysql.asp?filename=trysql_op_like) |
| NOT | Displays a record if the condition(s) is NOT TRUE | [Try it](https://www.w3schools.com/SQL/trysql.asp?filename=trysql_op_not) |
| OR | TRUE if any of the conditions separated by OR is TRUE | [Try it](https://www.w3schools.com/SQL/trysql.asp?filename=trysql_op_or) |
| SOME | TRUE if any of the subquery values meet the condition |  |
| = | Equal to | [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 to | [Try it](https://www.w3schools.com/SQL/trysql.asp?filename=trysql_op_greater_than2) |
| <= | Less than or equal to | [Try it](https://www.w3schools.com/SQL/trysql.asp?filename=trysql_op_less_than2) |
| <> | Not equal to |  |

DATA TYPES

|  |  |
| --- | --- |
| **Data type** | **Description** |
| CHARACTER(n) | Character string. Fixed-length n |
| VARCHAR(n) or CHARACTER VARYING(n) | Character string. Variable length. Maximum length n |
| BINARY(n) | Binary string. Fixed-length n |
| BOOLEAN | Stores TRUE or FALSE values |
| VARBINARY(n) or BINARY VARYING(n) | Binary string. Variable length. Maximum length n |
| INTEGER(p) | Integer numerical (no decimal). Precision p |
| SMALLINT | Integer numerical (no decimal). Precision 5 |
| INTEGER | Integer numerical (no decimal). Precision 10 |
| BIGINT | Integer numerical (no decimal). Precision 19 |
| DECIMAL(p,s) | Exact numerical, precision p, scale s. Example: decimal(5,2) is a number that has 3 digits before the decimal and 2 digits after the decimal |
| NUMERIC(p,s) | Exact numerical, precision p, scale s. (Same as DECIMAL) |
| FLOAT(p) | Approximate numerical, mantissa precision p. A floating number in base 10 exponential notation. The size argument for this type consists of a single number specifying the minimum precision |
| REAL | Approximate numerical, mantissa precision 7 |
| FLOAT | Approximate numerical, mantissa precision 16 |
| DOUBLE PRECISION | Approximate numerical, mantissa precision 16 |
| DATE | Stores year, month, and day values |
| TIME | Stores hour, minute, and second values |
| TIMESTAMP | Stores year, month, day, hour, minute, and second values |
| INTERVAL | Composed of a number of integer fields, representing a period of time, depending on the type of interval |
| ARRAY | A set-length and ordered collection of elements |
| MULTISET | A variable-length and unordered collection of elements |
| XML | Stores XML data |

# DB DATA TYPES

|  |  |  |
| --- | --- | --- |
| **Data type** | **Description** | **Storage** |
| char(n) | Fixed width character string. Maximum 8,000 characters | Defined width |
| varchar(n) | Variable width character string. Maximum 8,000 characters | 2 bytes + number of chars |
| varchar(max) | Variable width character string. Maximum 1,073,741,824 characters | 2 bytes + number of chars |
| text | Variable width character string. Maximum 2GB of text data | 4 bytes + number of chars |
| nchar | Fixed width Unicode string. Maximum 4,000 characters | Defined width x 2 |
| nvarchar | Variable width Unicode string. Maximum 4,000 characters |  |
| nvarchar(max) | Variable width Unicode string. Maximum 536,870,912 characters |  |
| ntext | Variable width Unicode string. Maximum 2GB of text data |  |
| bit | Allows 0, 1, or NULL |  |
| binary(n) | Fixed width binary string. Maximum 8,000 bytes |  |
| varbinary | Variable width binary string. Maximum 8,000 bytes |  |
| varbinary(max) | Variable width binary string. Maximum 2GB |  |
| image | Variable width binary string. Maximum 2GB |  |

# EXAMPLES

|  |  |
| --- | --- |
| **SQL Statement** | **Syntax** |
| AND / OR | SELECT column\_name(s) FROM table\_name WHERE condition AND|OR condition |
| ALTER TABLE | ALTER TABLE table\_name  ADD column\_name datatype  or  ALTER TABLE table\_name  DROP COLUMN column\_name |
| AS (alias) | SELECT column\_name AS column\_alias FROM table\_name  or  SELECT column\_name FROM table\_name  AS table\_alias |
| BETWEEN | SELECT column\_name(s) FROM table\_name WHERE column\_name BETWEEN value1 AND value2 |
| CREATE DATABASE | CREATE DATABASE database\_name |
| CREATE TABLE | CREATE TABLE table\_name ( column\_name1 data\_type, column\_name2 data\_type, column\_name3 data\_type, ... ) |
| CREATE INDEX | CREATE INDEX index\_name ON table\_name (column\_name)  or  CREATE UNIQUE INDEX index\_name ON table\_name (column\_name) |
| CREATE VIEW | CREATE VIEW view\_name AS SELECT column\_name(s) FROM table\_name WHERE condition |
| DELETE | DELETE FROM table\_name WHERE some\_column=some\_value  or  DELETE FROM table\_name  (**Note:**Deletes the entire table!!)  DELETE \* FROM table\_name  (**Note:**Deletes the entire table!!) |
| DROP DATABASE | DROP DATABASE database\_name |
| DROP INDEX | DROP INDEX table\_name.index\_name (SQL Server) DROP INDEX index\_name ON table\_name (MS Access) DROP INDEX index\_name (DB2/Oracle) ALTER TABLE table\_name DROP INDEX index\_name (MySQL) |
| DROP TABLE | DROP TABLE table\_name |
| EXISTS | IF EXISTS (SELECT \* FROM table\_name WHERE id = ?) BEGIN --do what needs to be done if exists END ELSE BEGIN --do what needs to be done if not END |
| GROUP BY | SELECT column\_name, aggregate\_function(column\_name) FROM table\_name WHERE column\_name operator value GROUP BY column\_name |
| HAVING | SELECT column\_name, aggregate\_function(column\_name) FROM table\_name WHERE column\_name operator value GROUP BY column\_name HAVING aggregate\_function(column\_name) operator value |
| IN | SELECT column\_name(s) FROM table\_name WHERE column\_name IN (value1,value2,..) |
| INSERT INTO | INSERT INTO table\_name VALUES (value1, value2, value3,....)  *or*  INSERT INTO table\_name (column1, column2, column3,...) VALUES (value1, value2, value3,....) |
| INNER JOIN | SELECT column\_name(s) FROM table\_name1 INNER JOIN table\_name2  ON table\_name1.column\_name=table\_name2.column\_name |
| LEFT JOIN | SELECT column\_name(s) FROM table\_name1 LEFT JOIN table\_name2  ON table\_name1.column\_name=table\_name2.column\_name |
| RIGHT JOIN | SELECT column\_name(s) FROM table\_name1 RIGHT JOIN table\_name2  ON table\_name1.column\_name=table\_name2.column\_name |
| FULL JOIN | SELECT column\_name(s) FROM table\_name1 FULL JOIN table\_name2  ON table\_name1.column\_name=table\_name2.column\_name |
| LIKE | SELECT column\_name(s) FROM table\_name WHERE column\_name LIKE pattern |
| ORDER BY | SELECT column\_name(s) FROM table\_name ORDER BY column\_name [ASC|DESC] |
| SELECT | SELECT column\_name(s) FROM table\_name |
| SELECT \* | SELECT \* FROM table\_name |
| SELECT DISTINCT | SELECT DISTINCT column\_name(s) FROM table\_name |
| SELECT INTO | SELECT \* INTO new\_table\_name [IN externaldatabase] FROM old\_table\_name  *or*  SELECT column\_name(s) INTO new\_table\_name [IN externaldatabase] FROM old\_table\_name |
| SELECT TOP | SELECT TOP number|percent column\_name(s) FROM table\_name |
| TRUNCATE TABLE | TRUNCATE TABLE table\_name |
| UNION | SELECT column\_name(s) FROM table\_name1 UNION SELECT column\_name(s) FROM table\_name2 |
| UNION ALL | SELECT column\_name(s) FROM table\_name1 UNION ALL SELECT column\_name(s) FROM table\_name2 |
| UPDATE | UPDATE table\_name SET column1=value, column2=value,... WHERE some\_column=some\_value |
| WHERE | SELECT column\_name(s) FROM table\_name WHERE column\_name operator value |