**SQL Concepts**

## **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 represents a 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 3 characters in length |
| WHERE ContactName LIKE 'a%o' | Finds any values that start with "a" and ends with "o" |

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

Example

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

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

Example

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

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

Example

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

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

Example

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

Or

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

## **BETWEEN with IN Example**

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

Example

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

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

Example

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 '01-July-1996' and '31-July-1996':

Example

SELECT \* FROM Orders WHERE OrderDate BETWEEN #01/07/1996# AND #31/07/1996#;

OR:

Example

SELECT \* FROM Orders  
WHERE OrderDate BETWEEN '1996-07-01' AND '1996-07-31';

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

Example

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

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

Example

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

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

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

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

Example

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

## **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

INNER JOIN Syntax

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

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

LEFT JOIN Syntax

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

RIGHT JOIN Syntax

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

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

FULL OUTER JOIN Syntax

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

SELECT *column\_name(s)* FROM *table1* FULL OUTER JOIN *table2* ON *table1.column\_name*=*table2.column\_name* WHERE *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 following SQL statement matches customers that are from the same city:**

Example

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

## **The SQL UNION Operator**

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

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

### UNION Syntax

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

UNION ALL Syntax

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

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

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

## **The SQL GROUP BY Statement**

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

Example

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

## **The SQL HAVING Clause**

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

### HAVING Syntax

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

The following SQL statement lists the number of customers in each country, sorted high to low (Only include countries with more than 5 customers):

Example

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

**The following SQL statement lists the employees that have registered more than 10 orders:**

Example

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;

**The following SQL statement lists if the employees "Davolio" or "Fuller" have registered more than 25 orders:**

Example

SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders  
FROM Orders INNER JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID WHERE LastName = 'Davolio' OR LastName = 'Fuller'  
GROUP BY LastName HAVING COUNT(Orders.OrderID) > 25;

## **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);

## **SQL EXISTS Examples**

The following SQL statement returns TRUE and lists the suppliers with a product price less than 20:

### Example

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

## **The SQL ANY and ALL Operators**

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

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

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

### ANY Syntax

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

ALL Syntax

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

## **SQL ANY Examples**

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

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

### Example

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

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

Example

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

## **SQL ALL Example**

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

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

### Example

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

## **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 oldtable WHERE condition;

Copy only some columns into a new table:

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

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

## **SQL SELECT INTO Examples**

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';

The following SQL statement copies data from more than one table into a new table:

SELECT Customers.CustomerName, Orders.OrderID  
INTO CustomersOrderBackup2017 FROM Customers  
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID;

**Tip:** 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;

## **The SQL INSERT INTO SELECT Statement**

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

INSERT INTO SELECT requires that data types in source and target tables match

The existing records in the target table are unaffected

INSERT INTO SELECT Syntax

Copy all columns from one table to another table:

* INSERT INTO *table2* SELECT \* FROM *table1*WHERE *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*;

## **SQL INSERT INTO SELECT Examples**

The following SQL statement copies "Suppliers" into "Customers" (the columns that are not filled with data, will contain NULL):

### Example

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

The following SQL statement copies only the German suppliers into "Customers":

Example

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

## **The SQL CASE Statement**

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

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:

### Example

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

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

Example

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

**SQL IFNULL(), ISNULL(), COALESCE(), and NVL() Functions**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **ProductName** | **UnitPrice** | **UnitsInStock** | **UnitsOnOrder** |
| 1 | Jarlsberg | 10.45 | 16 | 15 |
| 2 | Mascarpone | 32.56 | 23 |  |
| 3 | Gorgonzola | 15.67 | 9 | 20 |

Look at the following "Products" table: Suppose that the "UnitsOnOrder" column is optional, and may contain NULL values.

Look at the following SELECT statement:

SELECT ProductName, UnitPrice \* (UnitsInStock + UnitsOnOrder)  
FROM Products;

In the example above, if any of the "UnitsOnOrder" values are NULL, the result will be NULL.

## **Solutions**

**MySQL**

The MySQL [IFNULL()](https://www.w3schools.com/sql/func_mysql_ifnull.asp) function lets you return an alternative value if an expression is NULL:

SELECT ProductName, UnitPrice \* (UnitsInStock + IFNULL(UnitsOnOrder, 0))  
FROM Products;

or we can use the [COALESCE()](https://www.w3schools.com/sql/func_mysql_coalesce.asp) function, like this:

SELECT ProductName, UnitPrice \* (UnitsInStock + COALESCE(UnitsOnOrder, 0))  
FROM Products;

**SQL Server**

The SQL Server [ISNULL()](https://www.w3schools.com/sql/func_sqlserver_isnull.asp) function lets you return an alternative value when an expression is NULL:

SELECT ProductName, UnitPrice \* (UnitsInStock + ISNULL(UnitsOnOrder, 0))  
FROM Products;

**MS Access**

The MS Access [IsNull()](https://www.w3schools.com/sql/func_msaccess_isnull.asp) function returns TRUE (-1) if the expression is a null value, otherwise FALSE (0):

SELECT ProductName, UnitPrice \* (UnitsInStock + IIF(IsNull(UnitsOnOrder), 0, UnitsOnOrder))  
FROM Products;

**Oracle**

The Oracle NVL() function achieves the same result:

SELECT ProductName, UnitPrice \* (UnitsInStock + NVL(UnitsOnOrder, 0))  
FROM Products;

## **SQL Comments**

## **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;

## **Multi-line Comments**

Multi-line comments start with /\* and end with \*/.

Any text between /\* and \*/ will be ignored.

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

### Example

/\*Select all the columns  
of all the records  
in the Customers table:\*/  
SELECT \* FROM Customers;

## **The SQL CREATE DATABASE Statement**

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

### Syntax

CREATE DATABASE databasename;

## **The SQL DROP DATABASE Statement**

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

### Syntax

DROP DATABASE databasename;

## **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';

## **The SQL BACKUP WITH DIFFERENTIAL Statement**

A differential back up only backs up the parts of the database that have changed since the last full database backup.

### Syntax

BACKUP DATABASE databasename  
TO DISK = 'filepath'  
WITH DIFFERENTIAL;

Example

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

Example

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

## **SQL CREATE TABLE Example**

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

### Example

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

## **Create Table Using Another Table**

A copy of an existing table can also be created using CREATE TABLE.

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.

### Syntax

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

Example

CREATE TABLE TestTable AS  
SELECT customername, contactname  
FROM customers;

## **The SQL DROP TABLE Statement**

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

### Syntax

DROP TABLE table\_name;

## **SQL TRUNCATE TABLE**

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

### Syntax

TRUNCATE TABLE table\_name;

## **SQL ALTER TABLE Statement**

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

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

## **ALTER TABLE - ADD Column**

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

ALTER TABLE table\_name ADD column\_name datatype;

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

Example

ALTER TABLE Customers ADD Email varchar(255);

## **ALTER TABLE - DROP COLUMN**

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

ALTER TABLE table\_name DROP COLUMN column\_name;

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

Example

ALTER TABLE Customers DROP COLUMN Email;

## **ALTER TABLE - ALTER/MODIFY COLUMN**

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

**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*;

**Oracle 10G and later:**

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

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 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;

Notice that the "DateOfBirth" column is now of type year and is going to hold a year in a two- or four-digit format.

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

We use the following SQL statement:

ALTER TABLE Persons DROP COLUMN DateOfBirth;

## **SQL 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.

### Syntax

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

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

Example

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

## **SQL NOT NULL on ALTER TABLE**

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

ALTER TABLE Persons MODIFY Age int NOT NULL;

## **SQL UNIQUE Constraint**

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 table.

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

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

## **SQL PRIMARY KEY on CREATE TABLE**

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

**MySQL:**

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

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

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

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

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

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

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

## **SQL PRIMARY KEY on ALTER TABLE**

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

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

ALTER TABLE Persons ADD PRIMARY KEY (ID);

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

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

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

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

## **DROP a PRIMARY KEY Constraint**

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

**MySQL:**

ALTER TABLE Persons DROP PRIMARY KEY;

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

ALTER TABLE Persons DROP CONSTRAINT PK\_Person;

## **SQL FOREIGN KEY Constraint**

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

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

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

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

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

## **SQL FOREIGN KEY on CREATE TABLE**

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

**MySQL:**

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

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

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

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

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

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

## **SQL FOREIGN KEY on ALTER TABLE**

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

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

ALTER TABLE Orders ADD CONSTRAINT FK\_PersonOrder  
FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);

## **DROP a FOREIGN KEY Constraint**

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

**MySQL:**

ALTER TABLE Orders DROP FOREIGN KEY FK\_PersonOrder;

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

ALTER TABLE Orders DROP CONSTRAINT FK\_PersonOrder;

## **SQL CHECK Constraint**

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

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

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

## **SQL CHECK on CREATE TABLE**

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

**MySQL:**

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

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

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

To allow naming of a CHECK constraint, and for defining a CHECK 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,  
    City varchar(255),  
    CONSTRAINT CHK\_Person CHECK (Age>=18 AND City='Sandnes')  
);

## **SQL CHECK on ALTER TABLE**

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

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

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

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

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

ALTER TABLE Persons  
ADD CONSTRAINT CHK\_PersonAge CHECK (Age>=18 AND City='Sandnes');

## **DROP a CHECK Constraint**

To drop a CHECK constraint, use the following SQL:

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

ALTER TABLE Persons DROP CONSTRAINT CHK\_PersonAge;

**MySQL:**

ALTER TABLE Persons DROP CHECK CHK\_PersonAge;

## **SQL DEFAULT Constraint**

The DEFAULT constraint is used to provide a default value for a column. The default value will be added to all new records IF no other value is specified.

## **SQL DEFAULT on CREATE TABLE**

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

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

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

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

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

## **SQL DEFAULT on ALTER TABLE**

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

**MySQL:**

ALTER TABLE Persons ALTER City SET DEFAULT 'Sandnes';

**SQL Server:**

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

**MS Access:**

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

**Oracle:**

ALTER TABLE Persons MODIFY City DEFAULT 'Sandnes';

## **DROP a DEFAULT Constraint**

To drop a DEFAULT constraint, use the following SQL:

**MySQL:**

ALTER TABLE Persons ALTER City DROP DEFAULT;

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

ALTER TABLE Persons ALTER COLUMN City DROP DEFAULT;

## **SQL CREATE INDEX Statement**

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

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

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

### CREATE INDEX Syntax

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

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

### CREATE UNIQUE INDEX Syntax

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

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

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

## **CREATE INDEX Example**

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

CREATE INDEX idx\_lastname ON Persons (LastName);

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

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

## **DROP INDEX Statement**

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

**MS Access:**

DROP INDEX index\_name ON table\_name;

**SQL Server:**

DROP INDEX table\_name.index\_name;

**DB2/Oracle:**

DROP INDEX index\_name;

**MySQL:**

ALTER TABLE table\_name DROP INDEX index\_name;

## **AUTO INCREMENT Field**

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.

## **Syntax for MySQL**

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

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

MySQL uses the AUTO\_INCREMENT keyword to perform an auto-increment feature.

By default, the starting value for AUTO\_INCREMENT is 1, and it will increment by 1 for each new record.

To let the AUTO\_INCREMENT sequence start with another value, use the following SQL statement:

ALTER TABLE Persons AUTO\_INCREMENT=100;

To insert a new record into the "Persons" table, we will NOT have to specify a value for the "Personid" column (a unique value will be added automatically):

INSERT INTO Persons (FirstName,LastName) VALUES ('Lars','Monsen');

The SQL statement above would insert a new record into the "Persons" table. The "Personid" column would be assigned a unique value. The "FirstName" column would be set to "Lars" and the "LastName" column would be set to "Monsen".

## **Syntax for SQL Server**

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

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

The MS SQL Server uses the IDENTITY keyword to perform an auto-increment feature.

In the example above, the starting value for IDENTITY is 1, and it will increment by 1 for each new record.

**Tip:** To specify that the "Personid" column should start at value 10 and increment by 5, change it to IDENTITY(10,5).

To insert a new record into the "Persons" table, we will NOT have to specify a value for the "Personid" column (a unique value will be added automatically):

INSERT INTO Persons (FirstName,LastName) VALUES ('Lars','Monsen');

The SQL statement above would insert a new record into the "Persons" table. The "Personid" column would be assigned a unique value. The "FirstName" column would be set to "Lars" and the "LastName" column would be set to "Monsen".

## **Syntax for Access**

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

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

The MS Access uses the AUTOINCREMENT keyword to perform an auto-increment feature.

By default, the starting value for AUTOINCREMENT is 1, and it will increment by 1 for each new record.

**Tip:** To specify that the "Personid" column should start at value 10 and increment by 5, change the autoincrement to AUTOINCREMENT(10,5).

To insert a new record into the "Persons" table, we will NOT have to specify a value for the "Personid" column (a unique value will be added automatically):

INSERT INTO Persons (FirstName,LastName) VALUES ('Lars','Monsen');

The SQL statement above would insert a new record into the "Persons" table. The "Personid" column would be assigned a unique value. The "FirstName" column would be set to "Lars" and the "LastName" column would be set to "Monsen".

## **Syntax for Oracle**

In Oracle the code is a little bit more tricky.

You will have to create an auto-increment field with the sequence object (this object generates a number sequence).

Use the following CREATE SEQUENCE syntax:

CREATE SEQUENCE seq\_person  
MINVALUE 1  
START WITH 1  
INCREMENT BY 1  
CACHE 10;

The code above creates a sequence object called seq\_person, that starts with 1 and will increment by 1. It will also cache up to 10 values for performance. The cache option specifies how many sequence values will be stored in memory for faster access.

To insert a new record into the "Persons" table, we will have to use the nextval function (this function retrieves the next value from seq\_person sequence):

INSERT INTO Persons (Personid,FirstName,LastName)  
VALUES (seq\_person.nextval,'Lars','Monsen');

The SQL statement above would insert a new record into the "Persons" table. The "Personid" column would be assigned the next number from the seq\_person sequence. The "FirstName" column would be set to "Lars" and the "LastName" column would be set to "Monsen".

## **SQL Dates**

The most difficult part when working with dates is to be sure that the format of the date you are trying to insert, matches the format of the date column in the database.

As long as your data contains only the date portion, your queries will work as expected. However, if a time portion is involved, it gets more complicated.

## **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!

Now we want to select the records with an OrderDate of "2008-11-11" from the table above.

We use the following SELECT statement:

SELECT \* FROM Orders WHERE OrderDate='2008-11-11'

**Tip:** To keep your queries simple and easy to maintain, do not allow time components in your dates!

## **SQL CHECK on ALTER TABLE**

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

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

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

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

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

ALTER TABLE Persons   
ADD CONSTRAINT CHK\_PersonAge CHECK (Age>=18 AND City='Sandnes');

## **DROP a CHECK Constraint**

To drop a CHECK constraint, use the following SQL:

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

ALTER TABLE Persons DROP CONSTRAINT CHK\_PersonAge;

**MySQL:**

ALTER TABLE Persons DROP CHECK CHK\_PersonAge;

## **SQL CREATE VIEW Statement**

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

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

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

### CREATE VIEW Syntax

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

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

## **SQL CREATE VIEW Examples**

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

### Example

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

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);

## **SQL Updating a View**

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

### 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";

## **SQL Dropping a View**

A view is deleted with the DROP VIEW command.

### 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;

## **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();

## **SQL Hosting**

If you want your web site to be able to store and retrieve data from a database, your web server should have access to a database-system that uses the SQL language.

If your web server is hosted by an Internet Service Provider (ISP), you will have to look for SQL hosting plans. The most common SQL hosting databases are MS SQL Server, Oracle, MySQL, and MS Access.

## **MS SQL Server**

Microsoft's SQL Server is a popular database software for database-driven web sites with high traffic. SQL Server is a very powerful, robust and full featured SQL database system.

## **Oracle**

Oracle is also a popular database software for database-driven web sites with high traffic. Oracle is a very powerful, robust and full featured SQL database system.

## **MySQL**

MySQL is also a popular database software for web sites. MySQL is a very powerful, robust and full featured SQL database system. MySQL is an inexpensive alternative to the expensive Microsoft and Oracle solutions.

## **Access**

When a web site requires only a simple database, Microsoft Access can be a solution. Access is not well suited for very high-traffic, and not as powerful as MySQL, SQL Server, or Oracle.

**Advance SQL Concepts**

**Merge statement**

Perform Update,Isert and delete operation on a table using single SQL statement.

Avoid multiple insert, update and delete DML statements

MERGE INTO tablename

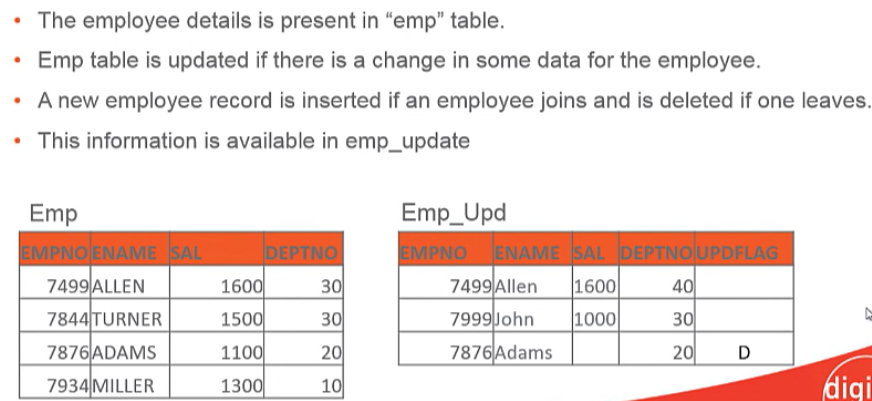
USING table\_reference name or a SELECT statement

ON(condition)

WHEN NOT Matched then Insert (col1[,col2…]) VALUES(value1[,value2…])

DELETE WHERE (delete\_condition);

Scenario -1



**Updateing the above table**

Allen is shifted to dept 40 and new employee john is joined and Adams is removed

MERGE INTO emp

USING EMP\_UPD

ON(emp.empno=emp\_upd.empno)

WHEN NOT Matched then Insert VALUES (emp\_upd.empno, emp\_upd.ename, emp\_upd.sal, emp\_upd.deptno)

WHEN MATCHED THEN UPDATE SET deptno = emp\_upd.deptno

DELETE WHERE emp\_upd.upd\_flag = ‘D’;

NOTE : we cannot update the same row of the target table multiple times in the same MERGE statement.

**Muilti table insert statement :**

INSERT ALL|FIRST

[WHEN condition THEN] INTO target [VALUES]

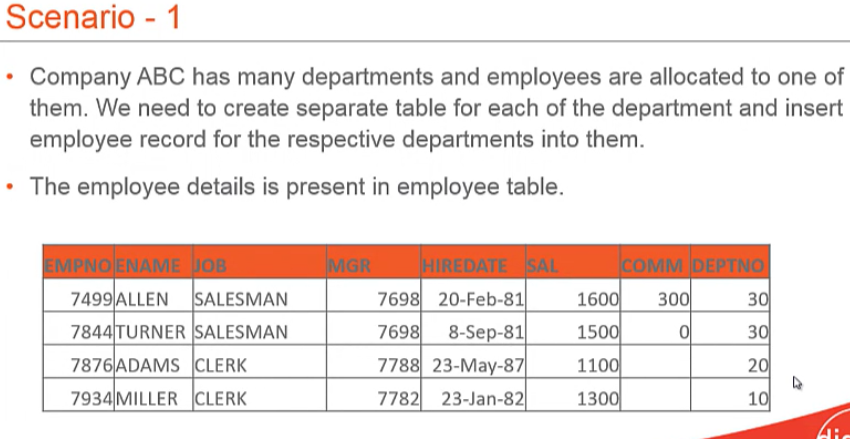
[WHEN condition THEN] INTO target [VALUES]

..

[ELSE] INTO target[VALUES]

Select …

FROM source\_query;



INSERT ALL

WHEN deptno=10 THEN INTO dep10

WHEN deptno=20 THEN INTO dep20

WHEN deptno=30 THEN INTO dep30

ELSE INTO dep\_default

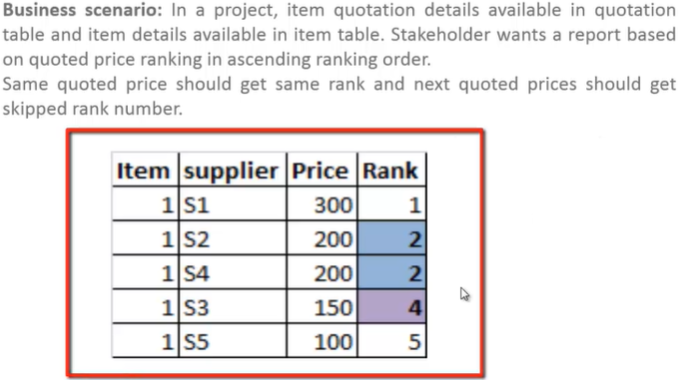
Select \* from emp;

NOTE :

Can perform multitable inserts only on tables not on views

**RANKS:** NORMAL RANK , DENSE RANK

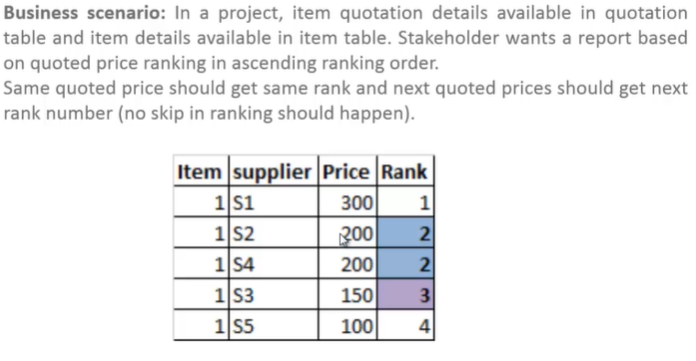
**NORMAL RANK:**



Select t1.supplierid,t1.itemcode,t2.description,t1.quotedprice,rank() over (partition by t1.itemcode order by t1.quotedprice desc) as rank from quotation t1

Inner join item t2 on t1.itemcode = t2.itemcode order by t1.itemcode

**DENSE RANK:**



Select t1.supplierid,t1.itemcode,t2.description,t1.quotedprice,DENSE\_rank() over (partition by t1.itemcode order by t1.quotedprice desc) as rank from quotation t1

Inner join item t2 on t1.itemcode = t2.itemcode order by t1.itemcode

**INSERT MINUS CASE QUERIES:**

**Intersect:** finding every source record is moved to target or not

**Scenario:** Emp and unit data has been transformed onto target DB using Lookup transformation. John wants to do data warehouse exhaustive validation on this transformed data.

**Source & Target Query:**

Select t1.EMpkey,t1.EmailID, t1.UnitKey, t2.Unitname, t1.Fname+’’+t1.Lname from [informatica\_sourceDB].[dbo].[EmpTable] t1 inner join [informatica\_sourceDB].[dbo].[Unittable] t2 on t1.Unitkey = t2.UnitKey

Intersect

Select Empkey,Emailid,Unitkey,UnitName,Name from [informatica\_targetDB].[dbo].[EmpUnitTable]

**Minus:** Finding extra records in the source compared to target and finding extra records there in target table compared to source

1. **Source to target**

Select t1.EMpkey,t1.EmailID, t1.UnitKey, t2.Unitname, t1.Fname+’’+t1.Lname from [informatica\_sourceDB].[dbo].[EmpTable] t1 inner join [informatica\_sourceDB].[dbo].[Unittable] t2 on t1.Unitkey = t2.UnitKey

except

Select Empkey,Emailid,Unitkey,UnitName,Name from [informatica\_targetDB].[dbo].[EmpUnitTable]

1. **target to source**

Select Empkey,Emailid,Unitkey,UnitName,Name from [informatica\_targetDB].[dbo].[EmpUnitTable]

except

Select t1.EMpkey,t1.EmailID, t1.UnitKey, t2.Unitname, t1.Fname+’’+t1.Lname from [informatica\_sourceDB].[dbo].[EmpTable] t1 inner join [informatica\_sourceDB].[dbo].[Unittable] t2 on t1.Unitkey = t2.UnitKey

**CASE Statement:**

**Scenario:** Emp and unit data has been transformed onto target DB using business logic which states that for all unit-1 employees will have unit name as F track, unit-2 will have M track and unit-3 will have N track. Carol wants to do data warehouse exhaustive validation on this transformed data.

Select t1.empkey,t1.emailid,t1.unitkey,

case

When t1.unitkey = 1 then ‘F track’

When t1.unitkey = 2 then ‘M track’

else ‘N track’

end

, t1.Fname+’’+t1.Lname from [informatica\_sourceDB].[dbo].[emptable] t1

**Pivot operations on table**

The PIVOT statement is used to convert table rows into columns, while the UNPIVOT operator converts columns back to rows. Reversing a PIVOT statement refers to the process of applying the  UNPIVOT operator to the already PIVOTED dataset in order to retrieve the original dataset.

In this article, we will study these three concepts on different examples.

**PIVOT Operator**

As mentioned earlier, the PIVOT operator converts table rows into columns. For example, if you have a table that looks like this:

## **Syntax:** The following syntax summarizes how to use the PIVOT operator.

SELECT <non-pivoted column>,

[first pivoted column] AS <column name>,

[second pivoted column] AS <column name>,

...

[last pivoted column] AS <column name>

FROM

(<SELECT query that produces the data>)

AS <alias for the source query>

PIVOT

(

<aggregation function>(<column being aggregated>)

FOR

[<column that contains the values that will become column headers>]

IN ( [first pivoted column], [second pivoted column],

... [last pivoted column])

) AS <alias for the pivot table>

<optional ORDER BY clause>;

## **Basic PIVOT Example :** The following code example produces a two-column table that has four rows.

USE AdventureWorks2014 ;

GO

SELECT DaysToManufacture, AVG(StandardCost) AS AverageCost

FROM Production.Product

GROUP BY DaysToManufacture;

Here is the result set.

DaysToManufacture AverageCost

----------------- -----------

0 5.0885

1 223.88

2 359.1082

4 949.4105

No products are defined with three DaysToManufacture.

The following code displays the same result, pivoted so that the DaysToManufacture values become the column headings. A column is provided for three [3] days, even though the results are NULL.

-- Pivot table with one row and five columns

SELECT 'AverageCost' AS Cost\_Sorted\_By\_Production\_Days,

[0], [1], [2], [3], [4]

FROM

(SELECT DaysToManufacture, StandardCost

FROM Production.Product) AS SourceTable

PIVOT

(

AVG(StandardCost)

FOR DaysToManufacture IN ([0], [1], [2], [3], [4])

) AS PivotTable;

Here is the result set.

Cost\_Sorted\_By\_Production\_Days 0 1 2 3 4

------------------------------ ----------- ----------- ----------- ----------- -----------

AverageCost 5.0885 223.88 359.1082 NULL 949.4105

## **Complex PIVOT Example**

A common scenario where PIVOT can be useful is when you want to generate cross-tabulation reports to give a summary of the data. For example, suppose you want to query the PurchaseOrderHeader table in the AdventureWorks2014 sample database to determine the number of purchase orders placed by certain employees. The following query provides this report, ordered by vendor.

USE AdventureWorks2014;

GO

SELECT VendorID, [250] AS Emp1, [251] AS Emp2, [256] AS Emp3, [257] AS Emp4, [260] AS Emp5

FROM

(SELECT PurchaseOrderID, EmployeeID, VendorID

FROM Purchasing.PurchaseOrderHeader) p

PIVOT

(

COUNT (PurchaseOrderID)

FOR EmployeeID IN

( [250], [251], [256], [257], [260] )

) AS pvt

ORDER BY pvt.VendorID;

Here is a partial result set.

VendorID Emp1 Emp2 Emp3 Emp4 Emp5

----------- ----------- ----------- ----------- ----------- -----------

1492 2 5 4 4 4

1494 2 5 4 5 4

1496 2 4 4 5 5

1498 2 5 4 4 4

1500 3 4 4 5 4

The results returned by this subselect statement are pivoted on the EmployeeID column.

SELECT PurchaseOrderID, EmployeeID, VendorID

FROM PurchaseOrderHeader;

The unique values returned by the EmployeeID column become fields in the final result set. As such, there's a column for each EmployeeID number specified in the pivot clause: in this case employees 164, 198, 223, 231, and 233. The PurchaseOrderID column serves as the value column, against which the columns returned in the final output, which are called the grouping columns, are grouped. In this case, the grouping columns are aggregated by the COUNTfunction. Notice that a warning message appears that indicates that any null values appearing in the PurchaseOrderID column weren't considered when computing the COUNT for each employee.

**Important**

When aggregate functions are used with PIVOT, the presence of any null values in the value column are not considered when computing an aggregation.

UNPIVOT carries out almost the reverse operation of PIVOT, by rotating columns into rows. Suppose the table produced in the previous example is stored in the database as pvt, and you want to rotate the column identifiers Emp1, Emp2, Emp3, Emp4, and Emp5 into row values that correspond to a particular vendor. As such, you must identify two additional columns. The column that will contain the column values that you're rotating (Emp1, Emp2,...) will be called Employee, and the column that will hold the values that currently exist under the columns being rotated will be called Orders. These columns correspond to the *pivot\_column*and *value\_column*, respectively, in the Transact-SQL definition. Here is the query.

-- Create the table and insert values as portrayed in the previous example.

CREATE TABLE pvt (VendorID int, Emp1 int, Emp2 int,

Emp3 int, Emp4 int, Emp5 int);

GO

INSERT INTO pvt VALUES (1,4,3,5,4,4);

INSERT INTO pvt VALUES (2,4,1,5,5,5);

INSERT INTO pvt VALUES (3,4,3,5,4,4);

INSERT INTO pvt VALUES (4,4,2,5,5,4);

INSERT INTO pvt VALUES (5,5,1,5,5,5);

GO

-- Unpivot the table.

SELECT VendorID, Employee, Orders

FROM

(SELECT VendorID, Emp1, Emp2, Emp3, Emp4, Emp5

FROM pvt) p

UNPIVOT

(Orders FOR Employee IN

(Emp1, Emp2, Emp3, Emp4, Emp5)

)AS unpvt;

GO

Here is a partial result set.

VendorID Employee Orders

----------- ----------- ------

1 Emp1 4

1 Emp2 3

1 Emp3 5

1 Emp4 4

1 Emp5 4

2 Emp1 4

2 Emp2 1

2 Emp3 5

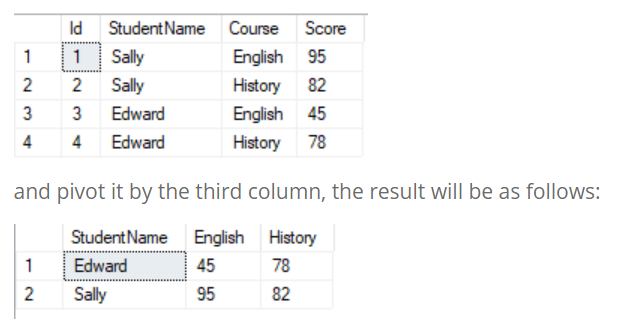
2 Emp4 5

2 Emp5 5

...

Notice that UNPIVOT isn't the exact reverse of PIVOT. PIVOT carries out an aggregation and merges possible multiple rows into a single row in the output. UNPIVOT doesn't reproduce the original table-valued expression result because rows have been merged. Also, null values in the input of UNPIVOT disappear in the output. When the values disappear, it shows that there may have been original null values in the input before the PIVOT operation.

The Sales.vSalesPersonSalesByFiscalYears view in the **AdventureWorks2012** sample database uses PIVOT to return the total sales for each salesperson, for each fiscal year. To script the view in SQL Server Management Studio, in **Object Explorer**, locate the view under the **Views** folder for the **AdventureWorks2012** database. Right-click the view name, and then select **Script View as**.



In the original table, we had two unique values for the Course columns – English and History. In the pivoted table, these unique values have been converted into columns. You can see that the score values for each new column remain unchanged. For instance, in the original table, a student, Sally, had scored 95 in English, unlike the values in the pivoted table.

CREATE DATABASE School

GO

USE School

GO

CREATE TABLE Students

(

Id INT PRIMARY KEY IDENTITY,

StudentName VARCHAR (50),

Course VARCHAR (50),

Score INT

)

GO

INSERT INTO Students VALUES ('Sally', 'English', 95 )

INSERT INTO Students VALUES ('Sally', 'History', 82)

INSERT INTO Students VALUES ('Edward', 'English', 45)

INSERT INTO Students VALUES ('Edward', 'History', 78)

The script above creates the School database. In the database, we create the Students table with four columns, such as Id, StudentName, Course, and Score. Finally, we add the four dummy records to the Students table.

Let’s PIVOT this table by the Course column. To do this, execute the following script:

SELECT \* FROM

(SELECT

StudentName,

Score,

Course

FROM

Students

)

AS StudentTable

PIVOT(

SUM(Score)

FOR Course IN ([English],[History])

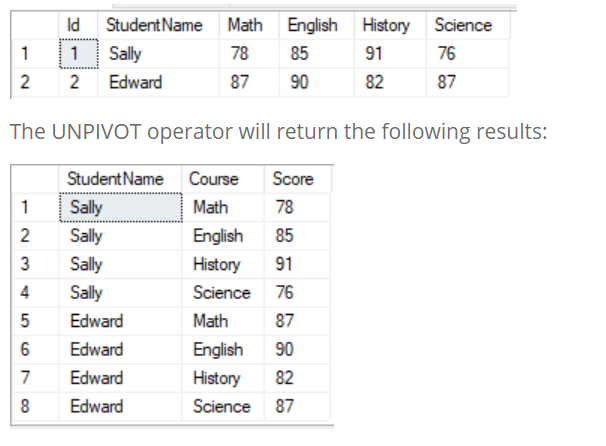
) AS SchoolPivot

Let’s see what is happening in the script. In the first line, we use the SELECT statement to define the columns that we want to add to the pivoted table. The first two columns are StudentName and Score. The data for these two columns will come directly from the Students table. The third column is Course. We want to PIVOT our table by the Course column, therefore, the Course column will be split into the number of columns equal to the values specified by the PIVOT operator for the Course column.

The syntax for the PIVOT operator is simple. First, you have to apply an aggregate function to the column the values of which you want to display in the pivoted columns. In our case, we want to show Score in the pivoted columns – English and History. Finally, we use a FOR statement to specify the pivot column and the unique values in it.

### UNPIVOT Operator

The UNPIVOT operator is used to convert table columns into rows. For instance, if you have a table that looks like this:



The columns of the original table have been converted to the rows in the unpivoted table. Let’s use that data to see how the UNPIVOT operator works in SQL.

To do this, execute the following script:

CREATE DATABASE School2

GO

USE School2

GO

CREATE TABLE Students

(

Id INT PRIMARY KEY IDENTITY,

StudentName VARCHAR (50),

Math INT,

English INT,

History INT,

Science INT

)

GO

INSERT INTO Students VALUES ('Sally', 78, 85, 91, 76 )

INSERT INTO Students VALUES ('Edward', 87, 90, 82, 87)

To apply the UNPIVOT operator to this table, run the following query:

SELECT StudentName, Course, Score

FROM Students

UNPIVOT

(

Score

FOR Course in (Math, English, History, Science)

) AS SchoolUnpivot

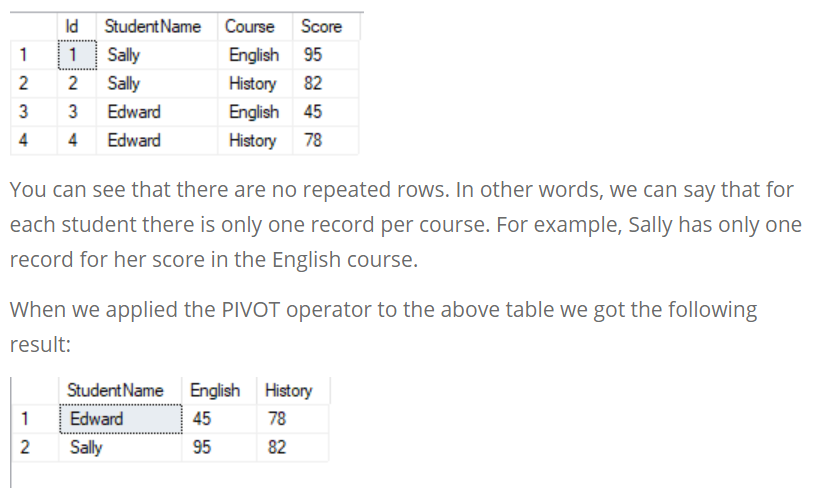
### Reversing a PIVOT

Reversing a PIVOT operator refers to the process of applying the UNPIVOT operator to a pivoted table in order to get back to the original table.

**Reversing Non-aggregate Pivoted Table**

Reversing a PIVOT operator is only possible if the pivoted table doesn’t contain aggregated data.

Let’s look at the table we used in the PIVOT section of this article.



Now, we are going to apply the UNPIVOT operator to this result and see if we can get back to the original table. To do this, execute the following script:

SELECT StudentName, Course, Score

FROM

(SELECT \* FROM

(SELECT

StudentName,

Score,

Course

FROM

Students

)

AS StudentTable

PIVOT(

SUM(Score)

FOR Course IN ([English],[History])

) AS SchoolPivot) PivotedResults

UNPIVOT

(

Score

FOR Course in (English, History)

) AS Schoolunpivot

Here we use a subquery to apply the UNPIVOT operator to the pivoted data. The inner query employs the PIVOT operator, while the outer query uses the UNPIVOT operator. At the output, you will see the original Students table.

**Reversing Aggregated Pivoted Table**

We said earlier that it is possible only to reverse a PIVOT operator that doesn’t contain aggregated data. Let’s try to reverse the PIVOT statement that contains aggregated data.

Add another record to the Students table of the School database that we created in the first section of this article. To do this, run the following query:

CREATE TABLE Students

(

Id INT PRIMARY KEY IDENTITY,

StudentName VARCHAR (50),

Course VARCHAR (50),

Score INT

)

GO

INSERT INTO Students VALUES ('Sally', 'English', 95 )

INSERT INTO Students VALUES ('Sally', 'History', 82)

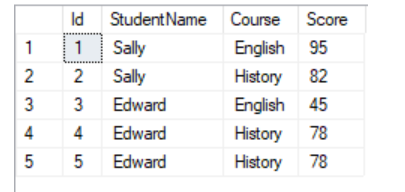
INSERT INTO Students VALUES ('Edward', 'English', 45)

INSERT INTO Students VALUES ('Edward', 'History', 78)

select \* from Students

INSERT INTO Students VALUES ('Edward', 'History', 78)

Now, if you select all the records from the Students table, you will get the following output:



We can see that we have a duplicate record for Edward’s score in History.

Now, apply the PIVOT operator to this table.

SELECT Id, StudentName, English, History

FROM Students

PIVOT

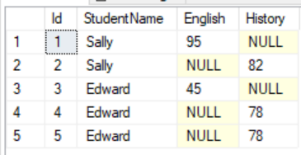
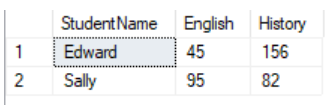
(

SUM (Score)

FOR Course in (English, History)

) AS Schoolpivot

Given Output Occurred Output



From the output, you can see that the SUM function in the PIVOT operator has added two scores to the History course taken by Edward. If you try to reverse the pivot of this table (i.e. apply the UNPIVOT operator), you will not receive the original table. It will return the four records instead of the original five. The History column for the student Edward will contain the aggregated result rather than the individual results.

To see this, execute the following script:

SELECT StudentName, Course, Score

FROM

(SELECT \* FROM

(SELECT

StudentName,

Score,

Course

FROM

Students

)

AS StudentTable

PIVOT(

SUM(Score)

FOR Course IN ([English],[History])

) AS SchoolPivot) PivotedResults

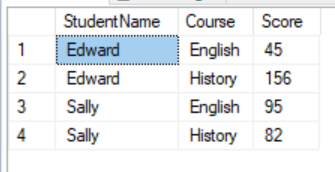
UNPIVOT

(

Score

FOR Course in (English, History)

) AS Schoolunpivot



Example ::

Need to count the different flags present in the columns and printing the result in single row

CREATE TABLE Pract\_Students

(

col INT

)

INSERT INTO Pract\_Students VALUES (1)

INSERT INTO Pract\_Students VALUES (-1)

INSERT INTO Pract\_Students VALUES (-1)

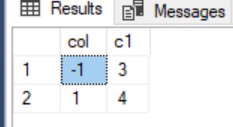
INSERT INTO Pract\_Students VALUES (1)

INSERT INTO Pract\_Students VALUES (1)

INSERT INTO Pract\_Students VALUES (-1)

INSERT INTO Pract\_Students VALUES (1)

select col,count(\*) as c1 from Pract\_Students group by col



select 1 as 'C1', -1 as 'C-1' from ( select col,count(\*) as c1 from Pract\_Students group by col) as s1 Pivot (sum(c1) for col in ([1] ,[-1]) ) as pivot1

