**SQL** is a database computer language designed for the retrieval and management of data in a relational database. **SQL** stands for **Structured Query Language**. This tutorial will give you a quick start to SQL. It covers most of the topics required for a basic understanding of SQL and to get a feel of how it works.

Why to Learn SQL?

SQL is Structured Query Language, which is a computer language for storing, manipulating and retrieving data stored in a relational database.

SQL is the standard language for Relational Database System. All the Relational Database Management Systems (RDMS) like MySQL, MS Access, Oracle, Sybase, Informix, Postgres and SQL Server use SQL as their standard database language.

Also, they are using different dialects, such as −

* MS SQL Server using T-SQL,
* Oracle using PL/SQL,
* MS Access version of SQL is called JET SQL (native format) etc.

Applications of SQL

As mentioned before, SQL is one of the most widely used query language over the databases. I'm going to list few of them here:

* Allows users to access data in the relational database management systems.
* Allows users to describe the data.
* Allows users to define the data in a database and manipulate that data.
* Allows to embed within other languages using SQL modules, libraries & pre-compilers.
* Allows users to create and drop databases and tables.
* Allows users to create view, stored procedure, functions in a database.
* Allows users to set permissions on tables, procedures and views.

## Audience

This **SQL tutorial** is prepared for beginners to help them understand the basic as well as the advanced concepts related to SQL languages. This tutorial will give you enough understanding on the various components of SQL along with suitable examples.

## Prerequisites

Before you start practicing with various types of examples given in this tutorial, I am assuming that you are already aware about what a database is, especially the RDBMS and what is a computer programming language.

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

## Syntax

The basic syntax of this CREATE DATABASE statement is as follows −

CREATE DATABASE DatabaseName;

Always the database name should be unique within the RDBMS.

## Example

If you want to create a new database <testDB>, then the CREATE DATABASE statement would be as shown below −

SQL> CREATE DATABASE testDB;

Make sure you have the admin privilege before creating any database. Once a database is created, you can check it in the list of databases as follows −

SQL> SHOW DATABASES;

+--------------------+

| Database |

+--------------------+

| information\_schema |

| AMROOD |

| TUTORIALSPOINT |

| mysql |

| orig |

| test |

| testDB |

+--------------------+

7 rows in set (0.00 sec)

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

## Syntax

The basic syntax of DROP DATABASE statement is as follows −

DROP DATABASE DatabaseName;

Always the database name should be unique within the RDBMS.

## Example

If you want to delete an existing database <testDB>, then the DROP DATABASE statement would be as shown below −

SQL> DROP DATABASE testDB;

**NOTE** − Be careful before using this operation because by deleting an existing database would result in loss of complete information stored in the database.

Make sure you have the admin privilege before dropping any database. Once a database is dropped, you can check it in the list of the databases as shown below −

SQL> SHOW DATABASES;

+--------------------+

| Database |

+--------------------+

| information\_schema |

| AMROOD |

| TUTORIALSPOINT |

| mysql |

| orig |

| test |

+--------------------+

6 rows in set (0.00 sec)

The SQL **USE** statement is used to select any existing database in the SQL schema.

## Syntax

The basic syntax of the USE statement is as shown below −

USE DatabaseName;

Always the database name should be unique within the RDBMS.

## Example

You can check the available databases as shown below −

SQL> SHOW DATABASES;

+--------------------+

| Database |

+--------------------+

| information\_schema |

| AMROOD |

| TUTORIALSPOINT |

| mysql |

| orig |

| test |

+--------------------+

6 rows in set (0.00 sec)

Now, if you want to work with the AMROOD database, then you can execute the following SQL command and start working with the AMROOD database.

SQL> USE AMROOD;

Creating a basic table involves naming the table and defining its columns and each column's data type.

The SQL **CREATE TABLE** statement is used to create a new table.

## Syntax

|  |  |  |
| --- | --- | --- |
| Eno | EName | ESalary |
| 101 | Abi | 10000 |
| 102 | Akash | 15000 |

The basic syntax of the CREATE TABLE statement is as follows −

CREATE TABLE table\_name(

column1 datatype,

column2 datatype,

column3 datatype,

.....

columnN datatype,

PRIMARY KEY( one or more columns )

);

CREATE TABLE is the keyword telling the database system what you want to do. In this case, you want to create a new table. The unique name or identifier for the table follows the CREATE TABLE statement.

Then in brackets comes the list defining each column in the table and what sort of data type it is. The syntax becomes clearer with the following example.

A copy of an existing table can be created using a combination of the CREATE TABLE statement and the SELECT statement. You can check the complete details at [Create Table Using another Table.](https://www.tutorialspoint.com/sql/sql-create-table-using-tables.htm)

## Example

The following code block is an example, which creates a CUSTOMERS table with an ID as a primary key and NOT NULL are the constraints showing that these fields cannot be NULL while creating records in this table −

SQL> CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25) ,

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

You can verify if your table has been created successfully by looking at the message displayed by the SQL server, otherwise you can use the **DESC** command as follows −

SQL> DESC CUSTOMERS;

+---------+---------------+------+-----+---------+-------+

| Field | Type | Null | Key | Default | Extra |

+---------+---------------+------+-----+---------+-------+

| ID | int(11) | NO | PRI | | |

| NAME | varchar(20) | NO | | | |

| AGE | int(11) | NO | | | |

| ADDRESS | char(25) | YES | | NULL | |

| SALARY | decimal(18,2) | YES | | NULL | |

+---------+---------------+------+-----+---------+-------+

5 rows in set (0.00 sec)

Now, you have CUSTOMERS table available in your database which you can use to store the required information related to customers.

The SQL **DROP TABLE** statement is used to remove a table definition and all the data, indexes, triggers, constraints and permission specifications for that table.

**NOTE** − You should be very careful while using this command because once a table is deleted then all the information available in that table will also be lost forever.

## Syntax

The basic syntax of this DROP TABLE statement is as follows −

DROP TABLE table\_name;

## Example

Let us first verify the CUSTOMERS table and then we will delete it from the database as shown below −

SQL> DESC CUSTOMERS;

+---------+---------------+------+-----+---------+-------+

| Field | Type | Null | Key | Default | Extra |

+---------+---------------+------+-----+---------+-------+

| ID | int(11) | NO | PRI | | |

| NAME | varchar(20) | NO | | | |

| AGE | int(11) | NO | | | |

| ADDRESS | char(25) | YES | | NULL | |

| SALARY | decimal(18,2) | YES | | NULL | |

+---------+---------------+------+-----+---------+-------+

5 rows in set (0.00 sec)

This means that the CUSTOMERS table is available in the database, so let us now drop it as shown below.

SQL> DROP TABLE CUSTOMERS;

Query OK, 0 rows affected (0.01 sec)

Now, if you would try the DESC command, then you will get the following error −

SQL> DESC CUSTOMERS;

ERROR 1146 (42S02): Table 'TEST.CUSTOMERS' doesn't exist

Here, TEST is the database name which we are using for our examples.

The SQL **INSERT INTO** Statement is used to add new rows of data to a table in the database.

### **Syntax**

There are two basic syntaxes of the INSERT INTO statement which are shown below.

INSERT INTO TABLE\_NAME (column1, column2, column3,...columnN)

VALUES (value1, value2, value3,...valueN);

Here, column1, column2, column3,...columnN are the names of the columns in the table into which you want to insert the data.

You may not need to specify the column(s) name in the SQL query if you are adding values for all the columns of the table. But make sure the order of the values is in the same order as the columns in the table.

The **SQL INSERT INTO** syntax will be as follows −

INSERT INTO TABLE\_NAME VALUES (value1,value2,value3,...valueN);

### **Example**

The following statements would create six records in the CUSTOMERS table.

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (6, 'Komal', 22, 'MP', 4500.00 );

You can create a record in the CUSTOMERS table by using the second syntax as shown below.

INSERT INTO CUSTOMERS

VALUES (7, 'Muffy', 24, 'Indore', 10000.00 );

All the above statements would produce the following records in the CUSTOMERS table as shown below.

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

## Populate one table using another table

You can populate the data into a table through the select statement over another table; provided the other table has a set of fields, which are required to populate the first table.

Here is the syntax −

INSERT INTO first\_table\_name [(column1, column2, ... columnN)]

SELECT column1, column2, ...columnN

FROM second\_table\_name

[WHERE condition];

INSERT INTO CUSTOMERS [(ID, NAME)]

SELECT ID, NAME FROM PURCHASE

WHERE (ID =101)

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

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

And a selection from the "Suppliers" table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **SupplierID** | **SupplierName** | **ContactName** | **Address** | **City** | **Postal Code** | **Country** |
| 1 | Exotic Liquid | Charlotte Cooper | 49 Gilbert St. | Londona | EC1 4SD | UK |
| 2 | New Orleans Cajun Delights | Shelley Burke | P.O. Box 78934 | New Orleans | 70117 | USA |
| 3 | Grandma Kelly's Homestead | Regina Murphy | 707 Oxford Rd. | Ann Arbor | 481 |  |

The SQL **SELECT** statement is used to fetch the data from a database table which returns this data in the form of a result table. These result tables are called result-sets.

## Syntax

The basic syntax of the SELECT statement is as follows −

SELECT column1, column2, columnN FROM table\_name;

Here, column1, column2... are the fields of a table whose values you want to fetch. If you want to fetch all the fields available in the field, then you can use the following syntax.

SELECT \* FROM table\_name;

## Example

SELECT \* FROM CUSTOMER;

Consider the CUSTOMERS table having the following records −

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

The following code is an example, which would fetch the ID, Name and Salary fields of the customers available in CUSTOMERS table.

SQL> SELECT ID, NAME, SALARY FROM CUSTOMERS;

This would produce the following result −

+----+----------+----------+

| ID | NAME | SALARY |

+----+----------+----------+

| 1 | Ramesh | 2000.00 |

| 2 | Khilan | 1500.00 |

| 3 | kaushik | 2000.00 |

| 4 | Chaitali | 6500.00 |

| 5 | Hardik | 8500.00 |

| 6 | Komal | 4500.00 |

| 7 | Muffy | 10000.00 |

+----+----------+----------+

If you want to fetch all the fields of the CUSTOMERS table, then you should use the following query.

SQL> SELECT \* FROM CUSTOMERS;

This would produce the result as shown below.

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | Delhi | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

SELECT NAME FROM CUSTOMERS WHERE ADDRESS = Delhi

Chaitali

Hardik

Komal

Muffy

The SQL **WHERE** clause is used to specify a condition while fetching the data from a single table or by joining with multiple tables. If the given condition is satisfied, then only it returns a specific value from the table. You should use the WHERE clause to filter the records and fetching only the necessary records.

The WHERE clause is not only used in the SELECT statement, but it is also used in the UPDATE, DELETE statement, etc., which we would examine in the subsequent chapters.

## Syntax

The basic syntax of the SELECT statement with the WHERE clause is as shown below.

SELECT column1, column2, columnN

FROM table\_name

WHERE [condition]

You can specify a condition using the [comparison or logical operators](https://www.tutorialspoint.com/sql/sql-operators.htm) like >, <, =, **LIKE, NOT**, etc. The following examples would make this concept clear.

## Example

Consider the CUSTOMERS table having the following records −

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Mumbai | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | MP | 10000.00 |

+----+----------+-----+-----------+----------+

The following code is an example which would fetch the ID, Name and Salary fields from the CUSTOMERS table, where the salary is greater than 2000 −

SQL> SELECT ID, NAME, SALARY

FROM CUSTOMERS

WHERE SALARY > 2000;

This would produce the following result −

+----+----------+----------+

| ID | NAME | SALARY |

+----+----------+----------+

| 4 | Chaitali | 6500.00 |

| 5 | Hardik | 8500.00 |

| 6 | Komal | 4500.00 |

| 7 | Muffy | 10000.00 |

+----+----------+----------+

The following query is an example, which would fetch the ID, Name and Salary fields from the CUSTOMERS table for a customer with the name **Hardik**.

Here, it is important to note that all the strings should be given inside single quotes (''). Whereas, numeric values should be given without any quote as in the above example.

SQL> SELECT ID, NAME, SALARY

FROM CUSTOMERS

WHERE NAME = 'Hardik';

This would produce the following result −

+----+----------+----------+

| ID | NAME | SALARY |

+----+----------+----------+

| 5 | Hardik | 8500.00 |

+----+----------+----------+

**The SQL SELECT DISTINCT Statement**

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

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

SELECT DISTINCT Syntax

SELECT DISTINCT column1, column2, ...

FROM table\_name;

SELECT DISTINCT ADDRESS FROM CUSTOMERS;

SELECT COUNT(DISTINCT Country) FROM Customers;

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | COUNTRY | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Mumbai | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | MP | 10000.00 |

+----+----------+-----+-----------+----------+

3

Operators in The WHERE Clause

The following operators can be used in the WHERE clause:

Operator Description

= Equal

> Greater than

< Less than

>= Greater than or equal

<= Less than or equal

<> Not equal. Note: In some versions of SQL this operator may be written as !=

BETWEEN Between a certain range

LIKE Search for a pattern

IN To specify multiple possible values for a column

EXAMPLE:

SELECT \* FROM Products

WHERE Price BETWEEN 50 AND 60;

**output:**

ProductID ProductName SupplierID CategoryID Unit Price

51 Manjimup Dried Apples 24 7 50 - 300 g pkgs. 53

59 Raclette Courdavault 28 4 5 kg pkg. 55

## ALTER TABLE - ADD Column

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

ALTER TABLE table\_name  
ADD column\_name datatype;

ALTER TABLE 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;

SQL ALTER TABLE Example

Look at the "Persons" table:

ID LastName FirstName Address City

1 Hansen Ola Timoteivn 10 Sandnes

2 Svendson Tove Borgvn 23 Sandnes

3 Pettersen Kari Storgt 20 Stavanger

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;

Notice that the new column, "DateOfBirth", is of type date and is going to hold a date. The data type specifies what type of data the column can hold. For a complete reference of all the data types available in MS Access, MySQL, and SQL Server, go to our complete Data Types reference.

The "Persons" table will now look like this:

ID LastName FirstName Address City DateOfBirth

1 Hansen Ola Timoteivn 10 Sandnes

2 Svendson Tove Borgvn 23 Sandnes

3 Pettersen Kari Storgt 20 Stavanger

ALTER TABLE Persons

DROP CITY;

Change Data Type Example

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.

DROP COLUMN Example

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

We use the following SQL statement:

ALTER TABLE Persons  
DROP COLUMN DateOfBirth;

The "Persons" table will now look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |

## TRUNCATE TABLE

The TRUNCATE TABLE command deletes the data inside a table, but not the table itself.

The following SQL truncates the table "Categories":

### **Example**

TRUNCATE TABLE Categories;

#### Relationship in DBMS

Any association between two entity types is called a relationship. Entities take part in the relationship. It is represented by a diamond shape.

***For example,*** A teacher teaches students. Here, "***teaches***" is a relationship and this is the relationship between a Teacher entity and a Student entity.

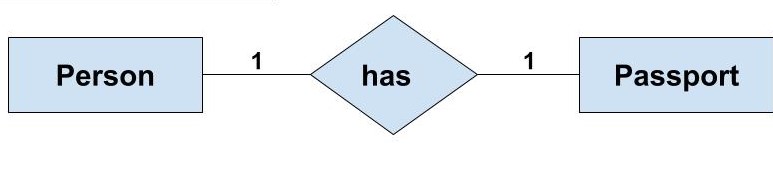
We have two entity types of 'Customer'(Customer\_id, Name, City, Phone) and 'Account'(Account\_no, Type, Balance). We store the data of 'Customer' in one table and his accounts details in the 'Account' table. Now, to link these two tables we need to insert the primary key 'Customer\_id' of the 'Customer' table in the 'Account' table. This key acts as a foreign key for the 'Account' table and refers to a column with the same name in the 'Customer' table. This is how a relationship between two tables is established. There are three types of relationships that can exist between two entities.

* One-to-One Relationship
* One-to-Many or Many-to-One Relationship
* Many-to-Many Relationship

#### One-to-One Relationship

Such a relationship exists when each record of one table is related to only one record of the other table.

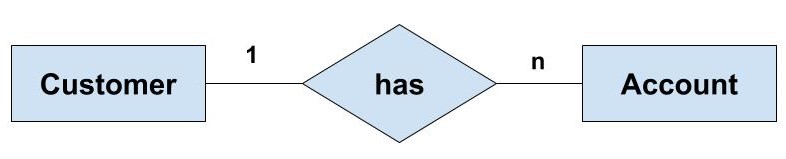
***For example,*** If there are two entities ‘Person’ (Id, Name, Age, Address)and ‘Passport’(Passport\_id, Passport\_no). So, each person can have only one passport and each passport belongs to only one person.



#### One-to-Many or Many-to-One Relationship

Such a relationship exists when each record of one table can be related to one or more than one record of the other table. This relationship is the most common relationship found. A one-to-many relationship can also be said as a many-to-one relationship depending upon the way we view it.

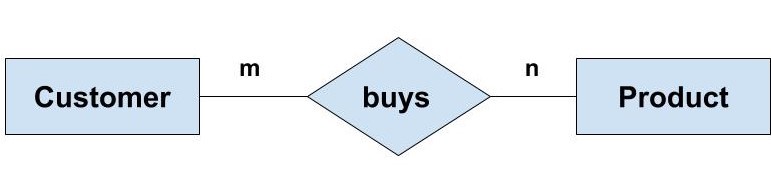
***For example,***If there are two entity type ‘Customer’ and ‘Account’ then each ‘Customer’ can have more than one ‘Account’ but each ‘Account’ is held by only one ‘Customer’. In this example, we can say that each Customer is associated with many Account. So, it is a one-to-many relationship. But, if we see it the other way i.e many Account is associated with one Customer then we can say that it is a many-to-one relationship.



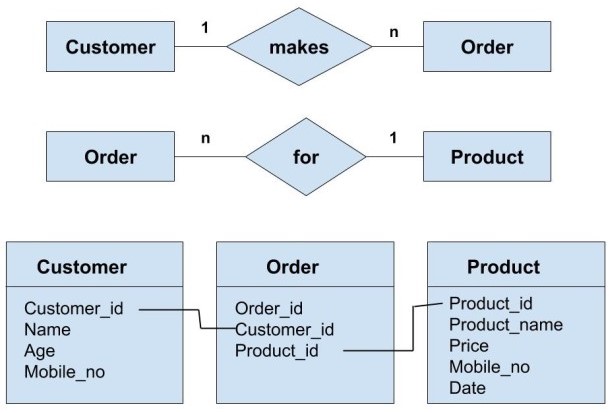
#### Many-to-Many Relationship

Such a relationship exists when each record of the first table can be related to one or more than one record of the second table and a single record of the second table can be related to one or more than one record of the first table. A many-to-many relationship can be seen as a two one-to-many relationship which is linked by a 'linking table' or 'associate table'. The linking table links two tables by having fields which are the primary key of the other two tables. We can understand this with the following example.

**Example:**If there are two entity type ‘Customer’ and ‘Product’ then each customer can buy more than one product and a product can be bought by many different customers.



Now, to understand the concept of the linking table here, we can have the ‘Order’ entity as a linking table which links the ‘Customer’ and ‘Product’ entity. We can break this many-to-many relationship in two one-to-many relationships. First, each ‘Customer’ can have many ‘Order’ whereas each ‘Order’ is related to only one ‘Customer’. Second, each ‘Order’ is related only one Product wheres there can many orders for the same Product.



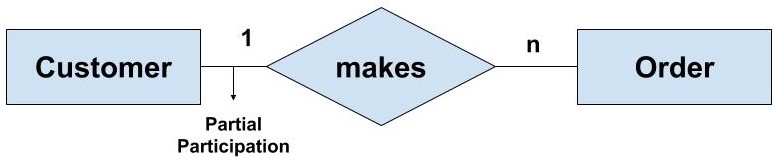
#### Participation Constraints

The relationship can be between two strong entity or a strong entity and a weak entity. Depending upon the type of entity participating in the relationship, the participation can be partial or total. There are two types of participation constraints:

* Partial Participation
* Total Participation

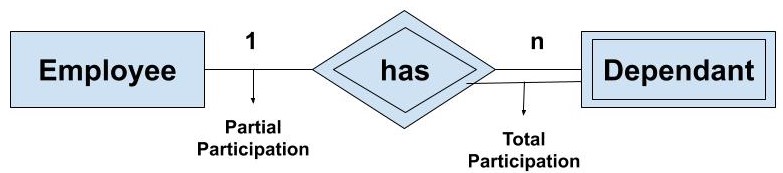
***Partial Participation:*** Partial Participation exists when all the entity of an entity type is not associated with one or the other entity of another entity type. This is represented by joining the relationship with the entity type with the help of one line.

***Example:*** We have two entity type ‘Customer’ and ‘Order’. Then there can be ‘Customer’ who have not done any order. So, here there is partial participation of the entity in the relationship.



***Total Participation:***Total Participation exists when all the entity of an entity type is associated with one or the other entity of another entity type. This is represented by joining the relationship with the entity type with the help of a double parallel line. Such a relationship usually exist between a strong entity and a weak entity.

***Example:*** We have two entity type ‘Employee’ and ‘Dependant’. Then all the ‘Dependent’ entity are related to one or the other ‘Employee’ entity. This is called total participation of the entity in the relationship. But, it may be possible that some ‘Employee’ is not related to any of the ‘Dependant’ entity. So, ‘Employee’ is showing partial participation whereas the ‘Dependant’ is showing total participation in the relationship.



## SQL FOREIGN KEY Constraint

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

A FOREIGN KEY is a field (or collection of fields) in one table, that refers to the [PRIMARY KEY](https://www.w3schools.com/sql/sql_primarykey.asp) in another table.

The table with the foreign key is called the child table, and the table with the primary key is called the referenced or parent table.

Look at the following two tables:

### **Persons Table**

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

### **Orders Table**

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

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

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

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

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

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;

## ADD CONSTRAINT

The ADD CONSTRAINT command is used to create a constraint after a table is already created.

The following SQL adds a constraint named "PK\_Person" that is a PRIMARY KEY constraint on multiple columns (ID and LastName):

### **Example**

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

**Example for update**

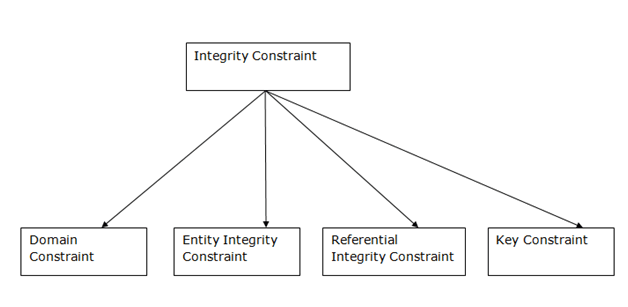
UPDATE Customers  
SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'  
WHERE CustomerID = 1;

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

# **Integrity Constraints**

* Integrity constraints are a set of rules. It is used to maintain the quality of information.
* Integrity constraints ensure that the data insertion, updating, and other processes have to be performed in such a way that data integrity is not affected.
* Thus, integrity constraint is used to guard against accidental damage to the database.

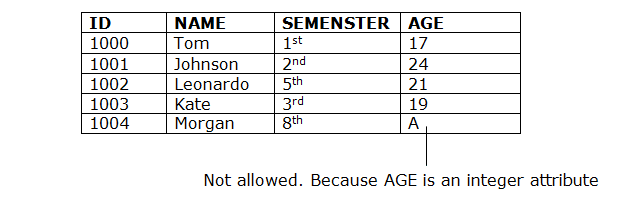
## Types of Integrity Constraint



### **1. Domain constraints**

* Domain constraints can be defined as the definition of a valid set of values for an attribute.
* The data type of domain includes string, character, integer, time, date, currency, etc. The value of the attribute must be available in the corresponding domain.

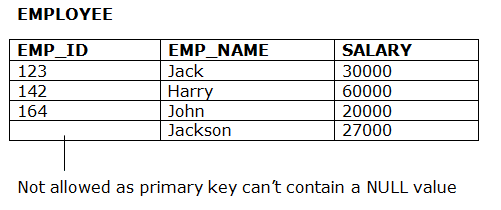
**Example:**



### **2. Entity integrity constraints**

* The entity integrity constraint states that primary key value can't be null.
* This is because the primary key value is used to identify individual rows in relation and if the primary key has a null value, then we can't identify those rows.
* A table can contain a null value other than the primary key field.

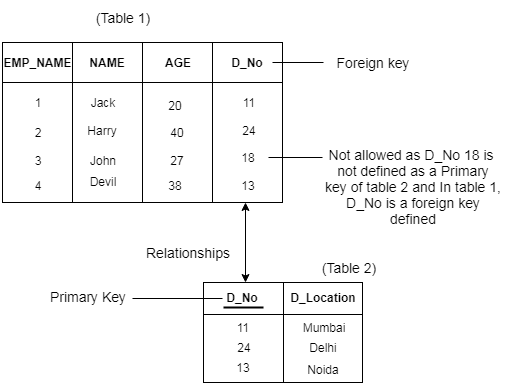
**Example:**



### **3. Referential Integrity Constraints**

* A referential integrity constraint is specified between two tables.
* In the Referential integrity constraints, if a foreign key in Table 1 refers to the Primary Key of Table 2, then every value of the Foreign Key in Table 1 must be null or be available in Table 2.

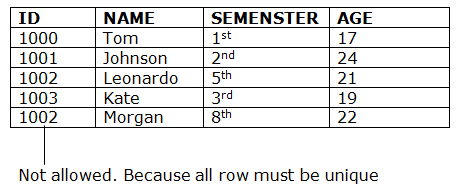
**Example:**



### **4. Key constraints**

* Keys are the entity set that is used to identify an entity within its entity set uniquely.
* An entity set can have multiple keys, but out of which one key will be the primary key. A primary key can contain a unique and null value in the relational table.

**Example:**



### **5) What are the subsets of SQL?**

The following are the four significant subsets of the SQL:

* **Data definition language (DDL):** It defines the data structure that consists of commands like CREATE, ALTER, DROP, etc.
* **Data manipulation language (DML):** It is used to manipulate existing data in the database. The commands in this category are SELECT, UPDATE, INSERT, etc.
* **Data control language (DCL):** It controls access to the data stored in the database. The commands in this category include GRANT and REVOKE.
* **Transaction Control Language (TCL):** It is used to deal with the transaction operations in the database. The commands in this category are COMMIT, ROLLBACK, SET TRANSACTION, SAVEPOINT, etc.

Q1 Using which language can a user request information from a database ?

* **Query**
* Relational
* Structural
* Compiler

Q2 Which one of the following is a procedural language ?

* Domain relational calculus
* Tuple relational calculus
* **Relational algebra**
* Query language

Q3 The\_\_\_\_\_ operation allows the combining of two relations by merging pairs of tuples, one from each relation, into a single tuple.

* Select
* **Join**
* Union
* Intersection

Q4 The result which operation contains all pairs of tuples from the two relations, regardless of whether their attribute values match.

* Join
* **Cartesian product**
* Intersection
* Set difference

**Q5:** Query all columns for all American cities in the **CITY** table with populations larger than 100000. The **CountryCode** for America is USA.

The **CITY** table is described as follows:



SELECT \*

FROM CITY

WHERE

COUNTRYCODE = 'USA'

AND POPULATION > 100000;

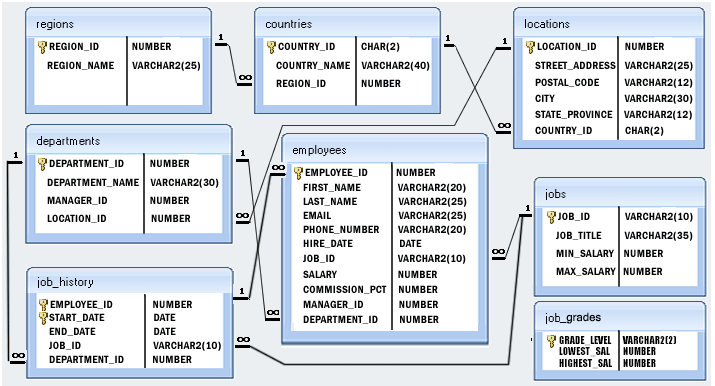
Q6: Query the **NAME** field for all American cities in the **CITY** table with populations larger than 120000. The CountryCode for America is USA.

The **CITY** table is described as follows:  


Create HR Database:

Create a table for **Employee, Job history, Department and Location**

**Select employee name from employee table where starting date is 1988**

1

​

**Q 7: Write a SQL statement to display a string "This is SQL Exercise, Practice and Solution".**

**select "This is SQL exercise,Practice and Solution"**

**This is SQL exercise,Practice and Solution**

**Q8: Write a query to display the result of an arithmetic expression.**

**Select 10 + 15 – 5 \* 2**

**Output: 15**

**Q 9: From the following table, write a SQL query to find the unique salespeople ID. Return salesman\_id.**

**ord\_no purch\_amt ord\_date customer\_id salesman\_id**

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

70001 150.5 2012-10-05 3005 5002

70009 270.65 2012-09-10 3001 5005

70002 65.26 2012-10-05 3002 5001

70004 110.5 2012-08-17 3009 5003

70007 948.5 2012-09-10 3005 5002

70005 2400.6 2012-07-27 3007 5001

70008 5760 2012-09-10 3002 5001

70010 1983.43 2012-10-10 3004 5006

70003 2480.4 2012-10-10 3009 5003

70012 250.45 2012-06-27 3008 5002

70011 75.29 2012-08-17 3003 5007

70013 3045.6 2012-04-25 3002 5001

SELECT DISTINCT salesman\_id FROM orders;

**Q 10:** Write a SQL query to show all details of the Prime Ministerial winners after 1972 of Menachem Begin and Yitzhak Rabin.

SELECT \* FROM nobel\_win

WHERE year >1972

AND winner IN ('Menachem Begin',

'Yitzhak Rabin');

YEAR SUBJECT WINNER COUNTRY CATEGORY

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

1970 Physics Hannes Alfven Sweden Scientist

1970 Physics Louis Neel France Scientist

1970 Chemistry Luis Federico Leloir France Scientist

1970 Physiology Ulf von Euler Sweden Scientist

1970 Physiology Bernard Katz Germany Scientist

1970 Literature Aleksandr Solzhenitsyn Russia Linguist

1970 Economics Paul Samuelson USA Economist

1970 Physiology Julius Axelrod USA Scientist

1971 Physics Dennis Gabor Hungary Scientist

1971 Chemistry Gerhard Herzberg Germany Scientist

1971 Peace Willy Brandt Germany Chancellor

1971 Literature Pablo Neruda Chile Linguist

1971 Economics Simon Kuznets Russia Economist

1978 Peace Anwar al-Sadat Egypt President

1978 Peace Menachem Begin Israel Prime Minister

1987 Chemistry Donald J. Cram USA Scientist

1987 Chemistry Jean-Marie Lehn France Scientist

1987 Physiology Susumu Tonegawa Japan Scientist

1994 Economics Reinhard Selten Germany Economist

1994 Peace Yitzhak Rabin Israel Prime Minister

1987 Physics Johannes Georg Bednorz Germany Scientist

1987 Literature Joseph Brodsky Russia Linguist

1987 Economics Robert Solow USA Economist

1994 Literature Kenzaburo Oe Japan Linguist

## SQL JOIN

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

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

|  |  |  |
| --- | --- | --- |
| **OrderID** | **CustomerID** | **OrderDate** |
| 10308 | 2 | 1996-09-18 |
| 10309 | 37 | 1996-09-19 |
| 10310 | 77 | 1996-09-20 |

Then, look at a selection from the "Customers" table:

|  |  |  |  |
| --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Country** |
| 1 | Alfreds Futterkiste | Maria Anders | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mexico |

Notice that the "CustomerID" column in the "Orders" table refers to the "CustomerID" in the "Customers" table. The relationship between the two tables above is the "CustomerID" column.

Then, we can create the following SQL statement (that contains an INNER JOIN), that selects records that have matching values in both tables:

### **Example**

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

and it will produce something like this:

|  |  |  |
| --- | --- | --- |
| **OrderID** | **CustomerName** | **OrderDate** |
| 10308 | Ana Trujillo Emparedados y helados | 9/18/1996 |
| 10365 | Antonio Moreno Taquería | 11/27/1996 |
| 10383 | Around the Horn | 12/16/1996 |
| 10355 | Around the Horn | 11/15/1996 |
| 10278 | Berglunds snabbköp | 8/12/1996 |

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

## Test Yourself With Exercises

## 1 Exercise:

Insert the missing parts in the JOIN clause to join the two tables Orders and Customers, using the CustomerID field in both tables as the relationship between the two tables.

SELECT \*

FROM Orders

LEFT JOIN Customers

=;

ON Orders.CustomerID = Customers.CustomerID;

# 2) **Exercise:**

Insert the missing parts in the JOIN clause to join the two tables Orders and Customers, using the CustomerID field in both tables as the relationship between the two tables.

SELECT \*

FROM Orders

LEFT JOIN Customers

=;

ON Orders.CustomerID = Customers.CustomerID;

## SQL INNER JOIN Keyword

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

### **INNER JOIN Syntax**

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



## Demo Database

Below is a selection from the "Orders" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OrderID** | **CustomerID** | **EmployeeID** | **OrderDate** | **ShipperID** |
| 10308 | 2 | 7 | 1996-09-18 | 3 |
| 10309 | 37 | 3 | 1996-09-19 | 1 |
| 10310 | 77 | 8 | 1996-09-20 | 2 |

And a selection from the "Customers" table:

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

## SQL INNER JOIN Example

The following SQL statement selects all orders with customer information:

### **Example**

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

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

## JOIN Three Tables

The following SQL statement selects all orders with customer and shipper information:

### **Example**

SELECT Orders.OrderID, Customers.CustomerName, Shippers.ShipperName  
FROM ((Orders  
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID)  
INNER JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID);

**3 Exercise :**

Choose the correct JOIN clause to select all records from the two tables where there is a match in both tables.

SELECT \*

FROM Orders

 INNER JOIN Customers

ON Orders.CustomerID=Customers.CustomerID;

## SQL LEFT JOIN Keyword

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

### **LEFT JOIN Syntax**

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

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



## Demo Database

well-known Northwind sample database.

Below is a selection from the "Customers" table:

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

And a selection from the "Orders" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OrderID** | **CustomerID** | **EmployeeID** | **OrderDate** | **ShipperID** |
| 10308 | 2 | 7 | 1996-09-18 | 3 |
| 10309 | 37 | 3 | 1996-09-19 | 1 |
| 10310 | 77 | 8 | 1996-09-20 | 2 |

## SQL LEFT JOIN Example

The following SQL statement will select all customers, and any orders they might have:

### **Example**

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

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

## SQL RIGHT JOIN Keyword

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

### **RIGHT JOIN Syntax**

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

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



## Demo Database

Below is a selection from the "Orders" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OrderID** | **CustomerID** | **EmployeeID** | **OrderDate** | **ShipperID** |
| 10308 | 2 | 7 | 1996-09-18 | 3 |
| 10309 | 37 | 3 | 1996-09-19 | 1 |
| 10310 | 77 | 8 | 1996-09-20 | 2 |

And a selection from the "Employees" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EmployeeID** | **LastName** | **FirstName** | **BirthDate** | **Photo** |
| 1 | Davolio | Nancy | 12/8/1968 | EmpID1.pic |
| 2 | Fuller | Andrew | 2/19/1952 | EmpID2.pic |
| 3 | Leverling | Janet | 8/30/1963 | EmpID3.pic |

## SQL RIGHT JOIN Example

The following SQL statement will return all employees, and any orders they might have placed:

### **Example**

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

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

|  |  |  |
| --- | --- | --- |
| **OrderID** | **LastName** | **FirstName** |
|  | West | Adam |
| 10248 | Buchanan | Steven |
| 10249 | Suyama | Michael |
| 10250 | Peacock | Margaret |
| 10251 | Leverling | Janet |
| 10252 | Peacock | Margaret |
| 10253 | Leverling | Janet |

## Exercise:

Choose the correct JOIN clause to select all the records from the Customers table plus all the matches in the Orders table.

SELECT \*

FROM Orders

 RIGHT JOIN Customers

ON Orders.CustomerID=Customers.CustomerID;

## SQL FULL OUTER JOIN Keyword

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

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

### **FULL OUTER JOIN Syntax**

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



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

## Demo Database

Below is a selection from the "Customers" table:

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

And a selection from the "Orders" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OrderID** | **CustomerID** | **EmployeeID** | **OrderDate** | **ShipperID** |
| 10308 | 2 | 7 | 1996-09-18 | 3 |
| 10309 | 37 | 3 | 1996-09-19 | 1 |
| 10310 | 77 | 8 | 1996-09-20 | 2 |

## SQL FULL OUTER JOIN Example

The following SQL statement selects all customers, and all orders:

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

A selection from the result set may look like this:

|  |  |
| --- | --- |
| **CustomerName** | **OrderID** |
| Null | 10309 |
| Null | 10310 |
| Alfreds Futterkiste | Null |
| Ana Trujillo Emparedados y helados | 10308 |
| Antonio Moreno Taquería | Null |

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

## SQL Self Join

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

### **Self Join Syntax**

SELECT column\_name(s)  
FROM table1 T1, table1 T2  
WHERE condition;

T1 and T2 are different table aliases for the same table.

## Demo Database

Below is a selection from the "Customers" table:

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

## SQL Self Join Example

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

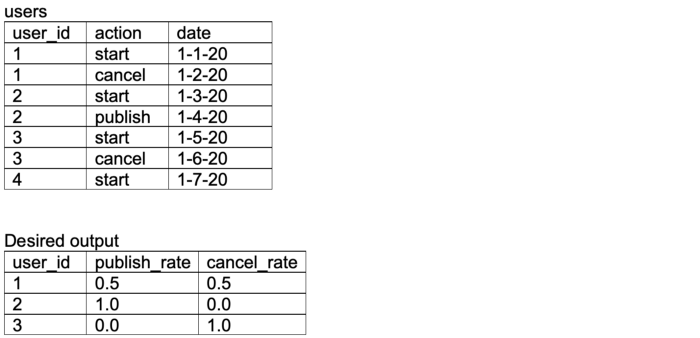
### **Example**

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

|  |  |  |
| --- | --- | --- |
| **CustomerName1** | **CustomerName2** | **City** |
| Alfreds Futterkiste | Heli Süßwaren GmbH & Co. KG | Berlin |
| Alfreds Futterkiste | Heli Süßwaren GmbH & Co. KG | Berlin |
| Heli Süßwaren GmbH & Co. KG | Alfreds Futterkiste | Berlin |
| Heli Süßwaren GmbH & Co. KG | Heli Süßwaren GmbH & Co. KG | Berlin |
| Heli Süßwaren GmbH & Co. KG | Alfreds Futterkiste | Berlin |
| Heli Süßwaren GmbH & Co. KG | Heli Süßwaren GmbH & Co. KG | Berlin |
| Cactus Comidas para llevar | Océano Atlántico Ltda. | Buenos Aires |
| Cactus Comidas para llevar | Rancho grande | Buenos Aires |
| Océano Atlántico Ltda. | Cactus Comidas para llevar | Buenos Aires |
| Océano Atlántico Ltda. | Rancho grande | Buenos Aires |
| Rancho grande | Cactus Comidas para llevar | Buenos Aires |
| Rancho grande | Océano Atlántico Ltda. | Buenos Aires |
| Nord-Ost-Fisch Handelsgesellschaft mbH | Nord-Ost-Fisch Handelsgesellschaft mbH | Cuxhaven |
| Nord-Ost-Fisch Handelsgesellschaft mbH | Nord-Ost-Fisch Handelsgesellschaft mbH | Cuxhaven |

## 1. Cancellation rates

From the following table of user IDs, actions, and dates, write a query to return the publication and cancellation rate for each user.



1.5\*(0/1)

WITH users (user\_id, action, date)   
AS (VALUES   
(1,'start', CAST('01-01-20' AS date)),   
(1,'cancel', CAST('01-02-20' AS date)),   
(2,'start', CAST('01-03-20' AS date)),   
(2,'publish', CAST('01-04-20' AS date)),   
(3,'start', CAST('01-05-20' AS date)),   
(3,'cancel', CAST('01-06-20' AS date)),   
(1,'start', CAST('01-07-20' AS date)),   
(1,'publish', CAST('01-08-20' AS date))),  
*-- retrieve count of starts, cancels, and publishes for each user*t1 AS (  
SELECT user\_id,   
sum(CASE WHEN action = 'start' THEN 1 ELSE 0 END) AS starts,   
sum(CASE WHEN action = 'cancel' THEN 1 ELSE 0 END) AS cancels,   
sum(CASE WHEN action = 'publish' THEN 1 ELSE 0 END) AS publishes  
FROM users  
GROUP BY 1  
ORDER BY 1)*-- calculate publication, cancelation rate for each user by dividing by number of starts, casting as float by multiplying by 1.0*SELECT user\_id, 1.0\*publishes/starts AS publish\_rate, 1.0\*cancels/starts AS cancel\_rate  
FROM t1

 What column names are displayed when this SQL command is executed?

**SHOW** **COLUMNS** **FROM** tbl\_name **LIKE** '%name';

a) suffixed with ‘name’  
b) prefixed with ‘name’  
c) suffixed with ‘%name’  
d) prefixed with ‘%name’  
View Answer

Answer: a  
Explanation: The wildcard ‘%’ is used to indicate that any number of characters can replace it. All column names that end in ‘name’ are displayed. Additional information of columns like type and size are listed.

For a given set of relation schemes, create tables and perform the following

Simple Queries, Simple Queries with Aggregate functions, Queries with Aggregate

functions (group by and having clause), Queries involving- Date Functions, String

Functions , Math Functions

Join Queries- Inner Join, Outer Join

Subqueries- With IN clause, With EXISTS clause

**What is Relational Model?**

**Relational Model (RM)** represents the database as a collection of relations. A relation is nothing but a table of values. Every row in the table represents a collection of related data values. These rows in the table denote a real-world entity or relationship.

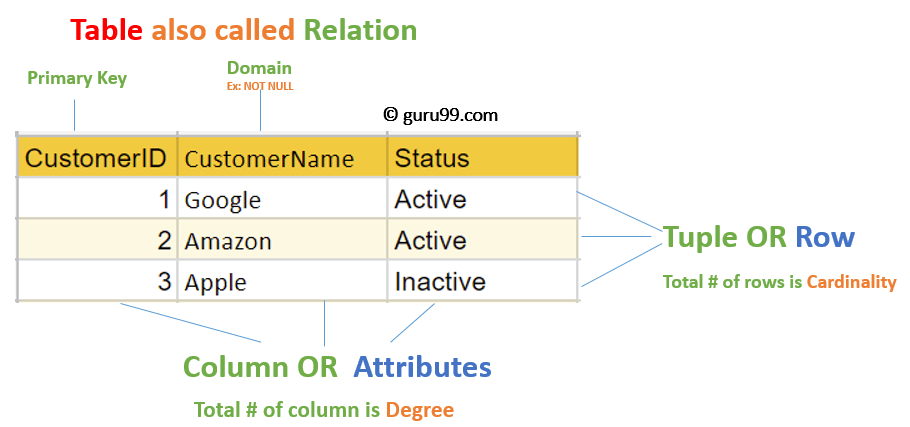
The table name and column names are helpful to interpret the meaning of values in each row. The data are represented as a set of relations. In the relational model, data are stored as tables. However, the physical storage of the data is independent of the way the data are logically organized.

**Some popular Relational Database management systems are:**

* DB2 and Informix Dynamic Server – IBM
* Oracle and RDB – Oracle
* SQL Server and Access – Microsoft

**Relational Model Concepts in DBMS**

1. **Attribute:** Each column in a Table. Attributes are the properties which define a relation. e.g., Student\_Rollno, NAME,etc.
2. **Tables** – In the Relational model the, relations are saved in the table format. It is stored along with its entities. A table has two properties rows and columns. Rows represent records and columns represent attributes.
3. **Tuple** – It is nothing but a single row of a table, which contains a single record.
4. **Relation Schema:** A relation schema represents the name of the relation with its attributes.
5. **Degree:** The total number of attributes which in the relation is called the degree of the relation.
6. **Cardinality:**Total number of rows present in the Table.
7. **Column:** The column represents the set of values for a specific attribute.
8. **Relation instance** – Relation instance is a finite set of tuples in the RDBMS system. Relation instances never have duplicate tuples.
9. **Relation key** – Every row has one, two or multiple attributes, which is called relation key.
10. **Attribute domain** – Every attribute has some pre-defined value and scope which is known as attribute domain



**Relational Integrity Constraints**

Relational Integrity constraints in DBMS are referred to conditions which must be present for a valid relation. These Relational constraints in DBMS are derived from the rules in the mini-world that the database represents.

There are many types of Integrity Constraints in DBMS. Constraints on the Relational database management system is mostly divided into three main categories are:

1. Domain Constraints
2. Key Constraints
3. Referential Integrity Constraints

**Operations in Relational Model**

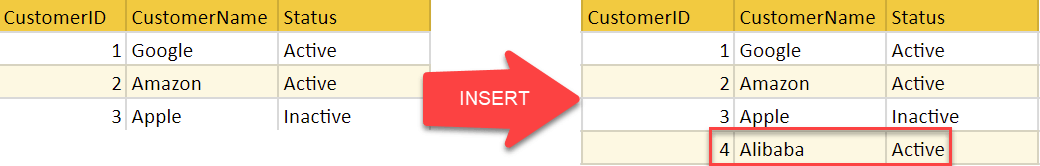
Four basic update operations performed on relational database model are

Insert, update, delete and select.

* Insert is used to insert data into the relation
* Delete is used to delete tuples from the table.
* Modify allows you to change the values of some attributes in existing tuples.
* Select allows you to choose a specific range of data.
* Whenever one of these operations are applied, integrity constraints specified on the relational database schema must never be violated.

### Insert Operation

* The insert operation gives values of the attribute for a new tuple which should be inserted into a relation.



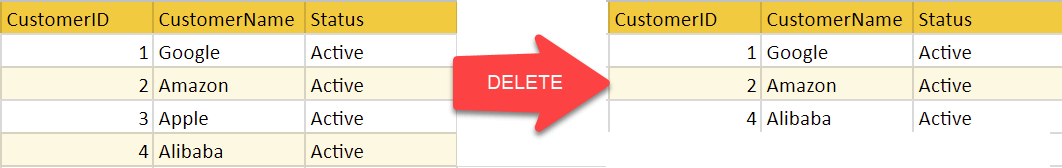
### Update Operation

* You can see that in the below-given relation table CustomerName= ‘Apple’ is updated from Inactive to Active.



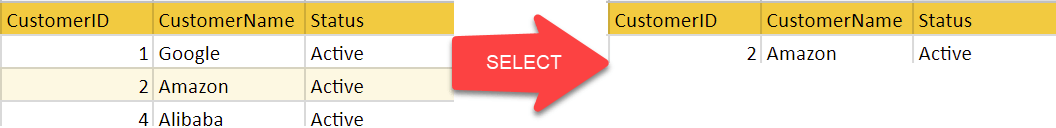
### Delete Operation

* To specify deletion, a condition on the attributes of the relation selects the tuple to be deleted.



* In the above-given example, CustomerName= “Apple” is deleted from the table.
* The Delete operation could violate referential integrity if the tuple which is deleted is referenced by foreign keys from other tuples in the same [database](https://www.guru99.com/introduction-to-database-sql.html).

### Select Operation



In the above-given example, CustomerName=”Amazon” is selected

## Best Practices for creating a Relational Model

* Data need to be represented as a collection of relations
* Each relation should be depicted clearly in the table
* Rows should contain data about instances of an entity
* Columns must contain data about attributes of the entity
* Cells of the table should hold a single value
* Each column should be given a unique name
* No two rows can be identical
* The values of an attribute should be from the same domain

## Advantages of Relational Database Model

* **Simplicity**: A Relational data model in DBMS is simpler than the hierarchical and network model.
* **Structural Independence**: The relational database is only concerned with data and not with a structure. This can improve the performance of the model.
* **Easy to use**: The Relational model in DBMS is easy as tables consisting of rows and columns are quite natural and simple to understand
* **Query capability**: It makes possible for a high-level query language like [SQL](https://www.guru99.com/sql.html) to avoid complex database navigation.
* **Data independence**: The Structure of Relational database can be changed without having to change any application.
* **Scalable**: Regarding a number of records, or rows, and the number of fields, a database should be enlarged to enhance its usability.

## Disadvantages of Relational Model

* Few relational databases have limits on field lengths which can’t be exceeded.
* Relational databases can sometimes become complex as the amount of data grows, and the relations between pieces of data become more complicated.
* Complex relational database systems may lead to isolated databases where the information cannot be shared from one system to another.

***INSERT DELETE AND UPDATE STATEMENT:***

**Sample table number one shows patient records and their basic information:**

**Patient table**

| **PATIENTID** | **SURNAME** | **FIRSTNAME** | **DOB** | **SEX** | **WEIGHT** | **HEIGHT** | **VACCINATED** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 15223 | Smith | Deniz | 12/31/2018 | F | 21.4 | 29.2 | Y |
| 15224 | Agarwal | Arjun | 08/29/2017 | M | 28.1 | 34.2 | Y |
| 15225 | Adams | Poppy | 02/14/2015 | F | 34.0 | 39.2 | N |
| 15226 | Johnson | Tierra | 08/15/2019 | F | 14.6 | 24.5 | Y |
| 15227 | Khouri | Mohammed | 03/30/2014 | M | 41.5 | 44.1 | Y |
| 15228 | Jones | Ben | 04/04/2011 | M | 70.1 | 52.2 | Y |
| 15229 | Kowalczyk | Alexandra | 08/27/2019 | F | 15.2 | 23.9 | Y |

**Sample table number two carries the different departments of the hospital and the groups and categories those departments fall under:**

**Department table**

| **DEPARTMENTID** | **DEPARTMENTNAME** | **GROUPNAME** | **CATEGORYNAME** |
| --- | --- | --- | --- |
| 1 | Cardiology | Heart Center | Clinical |
| 2 | Central ICU | Emergency | Clinical |
| 3 | Emergency | Emergency | Clinical |
| 4 | Communications | Administration | Operational |
| 5 | Oncology | Internal Medicine | Clinical |
| 6 | Neurology | Internal Medicine | Clinical |
| 7 | Human Resources | Administration | Operational |
| 8 | Pathology | Service | Technical |
| 9 | Radiology | Service | Technical |
| 10 | Pharmacy | Service | Technical |
| 11 | Executive Board | Administration | Operational |
| 12 | Urology | Surgery | Clinical |
| 13 | Hematology | Internal Medicine | Clinical |
| 14 | Montana Ward | Ward | Operational |
| 15 | Chicago Ward | Ward | Operational |
| 16 | Lincoln Ward | Ward | Operational |
| 17 | Yellowstone Ward | Ward | Operational |
| 18 | Brooklyn Ward | Ward | Operational |

**Sample table number three records patients’ visits over the years:**

**PatientAdmittance table**

| **PATIENTID** | **LASTADMITTED** | **LASTDISCHARGED** |
| --- | --- | --- |
| 33 | 12/29/1952 | 01/05/1953 |
| 34 | - | - |
| 35 | 08/01/2004 | 08/04/2004 |
| 36 | 07/28/2011 | 07/30/2011 |
| 37 | 05/27/1950 | 05/30/1950 |
| 38 | - | - |
| 39 | 10/11/1970 | 10/20/1970 |

Armed with these sample tables, let’s get into the nitty-gritty of INSERT, UPDATE, and DELETE.

### INSERT Data Using SQL

Let’s look at the basic structure of an SQL INSERT statement:

SQL

1 INSERT INTO tableName

2 (column1, column2, …)

3 VALUES (value1, value2, …)

Here’s how the INSERT statement would look:

SQL

1 INSERT INTO dbo.Patient

2 (Surname, FirstName, DOB, Sex, [Weight], Height, Vaccinated)

3 VALUES (NULL, NULL,'2019-11-19', 'F', 14.0, 23.1, 'No')

Inserting Multiple Rows

To insert more than one row of data with just one statement, use parentheses and commas to specify the distinct new rows.

SQL

1 INSERT INTO dbo.Patient

2 (Surname, FirstName, DOB, Sex, [Weight], Height, Vaccinated)

3 VALUES ('Hitson', 'George','2019-11-19', 'M', 13.9, 22.5, 'No'),

4 VALUES ('Hitson', 'Jenny','2019-11-19', 'F', 13.7, 22.3, 'No')

**Tip! Use a transaction to test out your insert without committing and permanently altering your table. To do this, start with BEGIN TRANSACTION, and end the transaction with either COMMIT, if you want to keep the changes, or ROLLBACK, if you want to reverse what you have done. Below is how we’d add a transaction to the above example:**

**Begin Transaction**

**SQL**

**1 BEGIN TRANSACTION**

**2 INSERT INTO dbo.Patient**

**3 (Surname, FirstName, DOB, Sex, [Weight], Height, Vaccinated)**

**4 VALUES ('Hitson', 'George','2019-11-19', 'M', 13.9, 22.5, 'No'),**

**5 ('Hitson', 'Jenny','2019-11-19', 'F', 13.7, 22.3, 'No')**

**6**

**7 ROLLBACK**

**8 –COMMIT**

**Inserting From One Table to Another**

Need to insert one or multiple rows of data from one table into another table? You can use the INSERT INTO SELECT statement.

Let’s say several of our young patients are taking part in a new drug trial, and you’re setting up a new table to record their participation. The basic structure of this statement is:

**SQL**

**1 INSERT INTO targetTable (column1, column2, …)**

**2 SELECT (column1, column2, …)**

**3 FROM sourceTable**

The statement for our drug trial example would look like this:

**SQL**

1 INSERT INTO [dbo].[DrugTrialAlpha]

2 (Surname, FirstName, DOB, Sex, DrugOrPlacebo, Notes)

3 SELECT

4 p.Surname, p.FirstName, p.DOB, p.Sex, 'Drug', NULL

5 FROM dbo.Patient AS p

6 WHERE

7 p.PatientID IN (15226, 15229, 15230)

8 UNION

9 SELECT

10 p.Surname, p.FirstName, p.DOB, p.Sex, 'Placebo', NULL

11 FROM dbo.Patient AS p

12 WHERE

13 p.PatientID IN (15231)

**You could also write the example as a Common Table Expression (CTE). Among other things, CTE can be used to create a temporary result set which can be reused during the query. Here’s how the same command written as a CTE would look:**

**SQL**

**1 WITH drugTrialPatients (Surname, FirstName, DOB, Sex, DrugOrPlacebo, Notes)**

**2 AS (**

**3 SELECT**

**4 p.Surname, p.FirstName, p.DOB, p.Sex, 'Drug', NULL**

**5 FROM dbo.Patient AS p**

**6 WHERE**

**7 p.PatientID IN (15226, 15229, 15230)**

**8 UNION**

**9 SELECT**

**10 p.Surname, p.FirstName, p.DOB, p.Sex, 'Placebo', NULL**

**11 FROM dbo.Patient AS p**

**12 WHERE**

**13 p.PatientID IN (15231)**

**14 )**

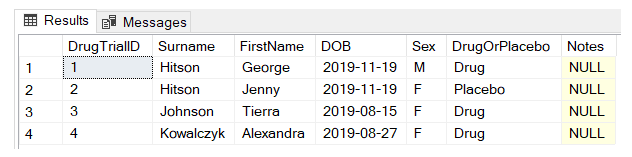
**15 INSERT INTO [dbo].[DrugTrialAlpha]**

**16 (Surname, FirstName, DOB, Sex, DrugOrPlacebo, Notes)**

**17 SELECT Surname, FirstName, DOB, Sex, DrugOrPlacebo, Notes**

**18 FROM drugTrialPatients**

**Table**



Tip! If you’re using SELECT to add data from one table to another, it’s a good idea as an initial step to run the SELECT statement alone—just to make sure the right rows are returned and you’re happy with the result!

**UPDATE Data Using SQL**

The SQL UPDATE statement is another common task used by data engineers and database administrators. Use it to change a column value for one or multiple rows.

**Here’s how an SQL UPDATE statement is generally structured:**

SQL

1 UPDATE tableName

2 SET column1=value1, column2=value2,...

3 WHERE filterColumn=filterValue

Here’s an example using our case study. Our Emergency department is rebranding to the “Trauma and Emergency Surgery” department, so its name needs changing in the database. To update this name, you could execute this statement:

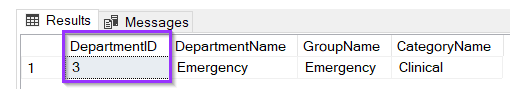
SQL

1 UPDATE dbo.Department

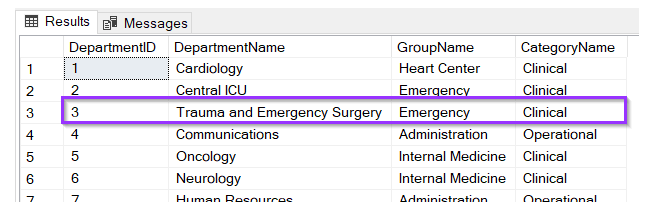
2 SET DepartmentName = 'Trauma and Emergency Surgery'

3 WHERE DepartmentID = 3

4 SELECT \* FROM dbo.Department WHERE DepartmentName = 'Emergency'



Here’s what the updated department name would look like in our table:



**Update Multiple Rows**

If you need to update multiple rows of data, it’s easy with the UPDATE statement. The WHERE clause is your friend here.

Imagine you want to recategorize all the inpatient wards from “Ward” to “Room” under the group column.

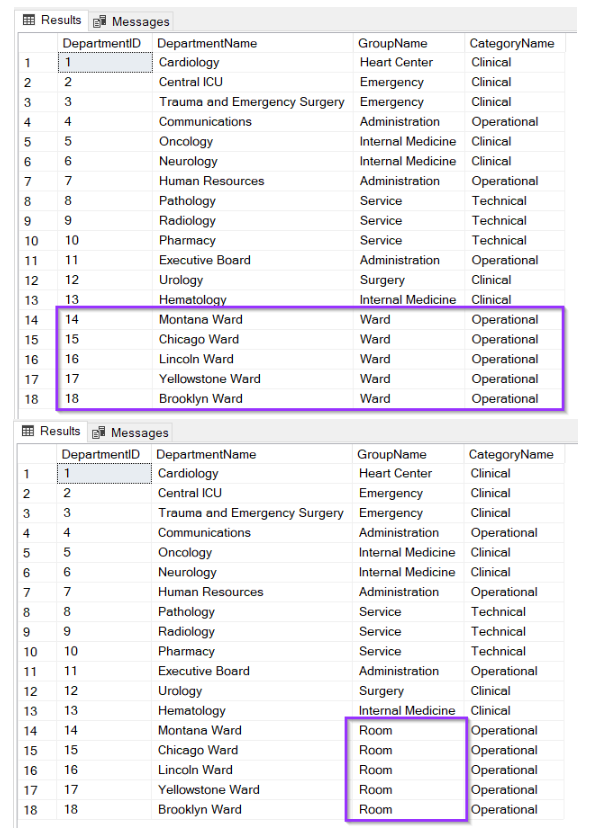
SQL

1 UPDATE dbo.Department

2 SET GroupName = 'Room'

3 WHERE GroupName = 'Ward'

Which would change the table to look like this:



### DELETE Data Using SQL

SQL DELETE is the diaper change of the SQL world. Is there something you don’t want in there? Delete it!

DELETE removes one or multiple rows from a table, but be careful! You need to make sure you know what you’re deleting before you go ahead and commit to the statement!

Here’s what your average SQL DELETE statement looks like:

SQL

1 DELETE tableName

2 WHERE filterColumn=filterValue;

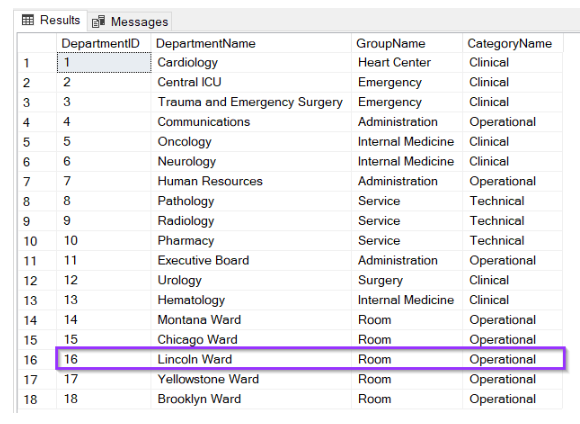
It’s just two parts: specifying the table and specifying WHERE—which rows you want to delete.

If you know the primary key of a row you want to delete, your job is simple. Do you want to delete the Lincoln Ward from your table? It’s just:

SQL

1 DELETE FROM dbo.Department

2 WHERE DepartmentID = 16



**Deleting Multiple Rows**

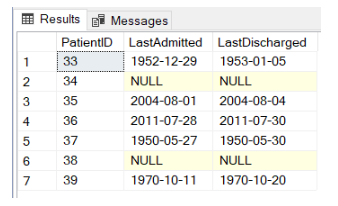
Back we go to Mickey Mouse Children’s Hospital. Let’s say the hospital’s data manager wants to delete all patient records of those who haven’t visited the hospital since 1969. Here’s how you could write that DELETE statement:

SQL

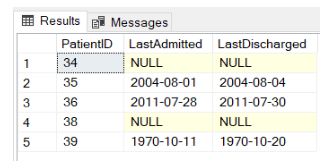
1 DELETE FROM dbo.PatientAdmittance

2 WHERE LastDischarged < '1969-01-01'

The result? This is the table before…



and this is how it looks after we run that script:



The best thing to do before running this DELETE statement, however, would be to test the result using SELECT. This will return all the rows you were about to delete, so you can check first whether you’re removing the right rows!

To run this test, you’d type:

SQL

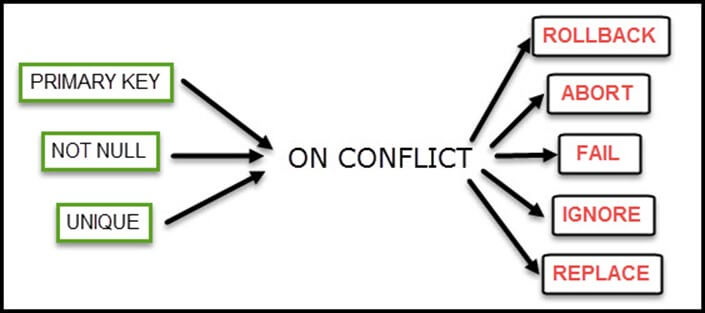
1 SELECT \* FROM dbo.PatientAdmittance

2 WHERE LastDischarged < '1969-01-01'

Warning! If you use DELETE without adding a WHERE clause, you’ll delete every row out of your table.

### How to declare the CONFLICT clause

You can declare the ON CONFLICT clause when you define a constraint for a column definition within the CREATE TABLE clause. Using the following syntax:



Numeric functions

## ABS(X)

The ABS() function returns the absolute value of X. Consider the following example −

SQL> SELECT ABS(2);

+---------------------------------------------------------+

| ABS(2) |

+---------------------------------------------------------+

| 2 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

SQL> SELECT ABS(-2);

+---------------------------------------------------------+

| ABS(2) |

+---------------------------------------------------------+

| 2 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## ACOS(X)

This function returns the arccosine of X. The value of X must range between -1 and 1 or NULL will be returned. Consider the following example −

SQL> SELECT ACOS(1);

+---------------------------------------------------------+

| ACOS(1) |

+---------------------------------------------------------+

| 0.000000 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## ASIN(X)

The ASIN() function returns the arcsine of X. The value of X must be in the range of -1 to 1 or NULL is returned.

SQL> SELECT ASIN(1);

+---------------------------------------------------------+

| ASIN(1) |

+---------------------------------------------------------+

| 1.5707963267949 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## ATAN(X)

This function returns the arctangent of X.

SQL> SELECT ATAN(1);

+---------------------------------------------------------+

| ATAN(1) |

+---------------------------------------------------------+

| 0.78539816339745 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## COS(X)

This function returns the cosine of X. The value of X is given in radians.

SQL>SELECT COS(90);

+---------------------------------------------------------+

| COS(90) |

+---------------------------------------------------------+

| -0.44807361612917 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## COT(X)

This function returns the cotangent of X. Consider the following example −

SQL>SELECT COT(1);

+---------------------------------------------------------+

| COT(1) |

+---------------------------------------------------------+

| 0.64209261593433 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## 

## DEGREES(X)

This function returns the value of X converted from radians to degrees.

SQL>SELECT DEGREES(PI());

+---------------------------------------------------------+

| DEGREES(PI()) |

+---------------------------------------------------------+

| 180.000000 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## EXP(X)

This function returns the value of e (the base of the natural logarithm) raised to the power of X.

SQL>SELECT EXP(3);

+---------------------------------------------------------+

| EXP(3) |

+---------------------------------------------------------+

| 20.085537 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## FLOOR(X)

This function returns the largest integer value that is not greater than X.

SQL>SELECT FLOOR(7.55);

+---------------------------------------------------------+

| FLOOR(7.55) |

+---------------------------------------------------------+

| 7 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## FORMAT(X,D)

The FORMAT() function is used to format the number X in the following format: ###,###,###.## truncated to D decimal places. The following example demonstrates the use and output of the FORMAT() function −

SQL>SELECT FORMAT(423423234.65434453,2);

+---------------------------------------------------------+

| FORMAT(423423234.65434453,2) |

+---------------------------------------------------------+

| 423,423,234.65 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## 

## GREATEST(n1,n2,n3,..........nn)

The GREATEST() function returns the greatest value in the set of input parameters (n1, n2, n3, a nd so on). The following example uses the GREATEST() function to return the largest number from a set of numeric values −

SQL>SELECT GREATEST(3,5,1,8,33,99,34,55,67,43);

+---------------------------------------------------------+

| GREATEST(3,5,1,8,33,99,34,55,67,43) |

+---------------------------------------------------------+

| 99 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## INTERVAL(N,N1,N2,N3,..........)

The INTERVAL() function compares the value of N to the value list (N1, N2, N3, and so on ). The function returns 0 if N < N1, 1 if N < N2, 2 if N <N3, and so on. It will return .1 if N is NULL. The value list must be in the form N1 < N2 < N3 in order to work properly. The following code is a simple example of how the INTERVAL() function works −

SQL>SELECT INTERVAL(6,1,2,3,4,5,6,7,8,9,10);

+---------------------------------------------------------+

| INTERVAL(6,1,2,3,4,5,6,7,8,9,10) |

+---------------------------------------------------------+

| 6 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## MOD(N,M)

This function returns the remainder of N divided by M. Consider the following example −

SQL>SELECT MOD(29,3);

+---------------------------------------------------------+

| MOD(29,3) |

+---------------------------------------------------------+

| 2 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## OCT(N)

The OCT() function returns the string representation of the octal number N. This is equivalent to using CONV(N,10,8).

SQL>SELECT OCT(12);

+---------------------------------------------------------+

| OCT(12) |

+---------------------------------------------------------+

| 14 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## ROUND(X)

## ROUND(X,D)

This function returns X rounded to the nearest integer. If a second argument, D, is supplied, then the function returns X rounded to D decimal places. D must be positive or all digits to the right of the decimal point will be removed. Consider the following example −

SQL>SELECT ROUND(5.693893);

+---------------------------------------------------------+

| ROUND(5.693893) |

+---------------------------------------------------------+

| 6 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

SQL>SELECT ROUND(5.693893,2);

+---------------------------------------------------------+

| ROUND(5.693893,2) |

+---------------------------------------------------------+

| 5.69 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## SIGN(X)

This function returns the sign of X (negative, zero, or positive) as -1, 0, or 1.

SQL>SELECT SIGN(-4.65);

+---------------------------------------------------------+

| SIGN(-4.65) |

+---------------------------------------------------------+

| -1 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

SQL>SELECT SIGN(0);

+---------------------------------------------------------+

| SIGN(0) |

+---------------------------------------------------------+

| 0 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

SQL>SELECT SIGN(4.65);

+---------------------------------------------------------+

| SIGN(4.65) |

+---------------------------------------------------------+

| 1 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## SIN(X)

This function returns the sine of X. Consider the following example −

SQL>SELECT SIN(90);

+---------------------------------------------------------+

| SIN(90) |

+---------------------------------------------------------+

| 0.893997 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## SQRT(X)

This function returns the non-negative square root of X. Consider the following example −

SQL>SELECT SQRT(49);

+---------------------------------------------------------+

| SQRT(49) |

+---------------------------------------------------------+

| 7 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## TRUNCATE(X,D)

This function is used to return the value of X truncated to D number of decimal places. If D is 0, then the decimal point is removed. If D is negative, then D number of values in the integer part of the value is truncated. Consider the following example −

SQL>SELECT TRUNCATE(7.536432,2);

+---------------------------------------------------------+

| TRUNCATE(7.536432,2) |

+---------------------------------------------------------+

| 7.53 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

TRUNCATE(7.536432,4)

7.5364

## Date functions:

## ADDDATE(date,INTERVAL expr unit), ADDDATE(expr,days)

When invoked with the INTERVAL form of the second argument, ADDDATE() is a synonym for DATE\_ADD(). The related function SUBDATE() is a synonym for DATE\_SUB(). For information on the INTERVAL unit argument, see the discussion for DATE\_ADD().

mysql> SELECT DATE\_ADD('1998-01-02', INTERVAL 31 DAY);

+---------------------------------------------------------+

| DATE\_ADD('1998-01-02', INTERVAL 31 DAY) |

+---------------------------------------------------------+

| 1998-02-02 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

mysql> SELECT ADDDATE('1998-01-02', INTERVAL 31 DAY);

+---------------------------------------------------------+

| ADDDATE('1998-01-02', INTERVAL 31 DAY) |

+---------------------------------------------------------+

| 1998-02-02 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

When invoked with the days form of the second argument, MySQL treats it as an integer number of days to be added to expr.

mysql> SELECT ADDDATE('1998-01-02', 31);

+---------------------------------------------------------+

| DATE\_ADD('1998-01-02', INTERVAL 31 DAY) |

+---------------------------------------------------------+

| 1998-02-02 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## ADDTIME(expr1,expr2)

ADDTIME() adds expr2 to expr1 and returns the result. The expr1 is a time or datetime expression, while the expr2 is a time expression.

mysql> SELECT ADDTIME('1997-12-31 23:59:59.999999','1 1:1:1.000002');

+---------------------------------------------------------+

| DATE\_ADD('1997-12-31 23:59:59.999999','1 1:1:1.000002') |

+---------------------------------------------------------+

| 1998-01-02 01:01:01.000001 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## CURRENT\_DATE and CURRENT\_DATE()

CURRENT\_DATE and CURRENT\_DATE() are synonyms for CURDATE()

## CURTIME()

Returns the current time as a value in 'HH:MM:SS' or HHMMSS format, depending on whether the function is used in a string or in a numeric context. The value is expressed in the current time zone.

mysql> SELECT CURTIME();

+---------------------------------------------------------+

| CURTIME() |

+---------------------------------------------------------+

| 23:50:26 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

mysql> SELECT CURTIME() + 0;

+---------------------------------------------------------+

| CURTIME() + 0 |

+---------------------------------------------------------+

| 235026 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## CURRENT\_TIME and CURRENT\_TIME()

CURRENT\_TIME and CURRENT\_TIME() are synonyms for CURTIME().

## CURRENT\_TIMESTAMP and CURRENT\_TIMESTAMP()

CURRENT\_TIMESTAMP and CURRENT\_TIMESTAMP() are synonyms for NOW().

## DATE(expr)

Extracts the date part of the date or datetime expression expr.

mysql> SELECT DATE('2003-12-31 01:02:03');

+---------------------------------------------------------+

| DATE('2003-12-31 01:02:03') |

+---------------------------------------------------------+

| 2003-12-31 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## DATE\_FORMAT(date,format)

This command formats the date value as per the format string.

The following specifiers may be used in the format string. The '%' character is required before the format specifier characters.

|  |  |
| --- | --- |
| **Sr.No.** | **Specifier & Description** |
| 1 | **%a**  Abbreviated weekday name (Sun..Sat) |
| 2 | **%b**  Abbreviated month name (Jan..Dec) |
| 3 | **%c**  Month, numeric (0..12) |
| 4 | **%D**  Day of the month with English suffix (0th, 1st, 2nd, 3rd, .) |
| 5 | **%d**  Day of the month, numeric (00..31) |
| 6 | **%e**  Day of the month, numeric (0..31) |
| 7 | **%f**  Microseconds (000000..999999) |
| 8 | **%H**  Hour (00..23) |
| 9 | **%h**  Hour (01..12) |
| 10 | **%I**  Hour (01..12) |
| 11 | **%i**  Minutes, numeric (00..59) |
| 12 | **%j**  Day of year (001..366) |
| 13 | **%k**  Hour (0..23) |
| 14 | **%l**  Hour (1..12) |
| 15 | **%M**  Month name (January..December) |
| 16 | **%m**  Month, numeric (00..12) |
| 17 | **%p**  AM or PM |
| 18 | **%r**  Time, 12-hour (hh:mm:ss followed by AM or PM) |
| 19 | **%S**  Seconds (00..59) |
| 20 | **%s**  Seconds (00..59) |
| 21 | **%T**  Time, 24-hour (hh:mm:ss) |
| 22 | **%U**  Week (00..53), where Sunday is the first day of the week |
| 23 | **%u**  Week (00..53), where Monday is the first day of the week |
| 24 | **%V**  Week (01..53), where Sunday is the first day of the week; used with %X |
| 25 | **%v**  Week (01..53), where Monday is the first day of the week; used with %x |
| 26 | **%W**  Weekday name (Sunday..Saturday) |
| 27 | **%w**  Day of the week (0=Sunday..6=Saturday) |
| 28 | **%X**  Year for the week where Sunday is the first day of the week, numeric, four digits; used with %V |
| 29 | **%x**  Year for the week, where Monday is the first day of the week, numeric, four digits; used with %v |
| 30 | **%Y**  Year, numeric, four digits |
| 31 | **%y**  Year, numeric (two digits) |
| 32 | **%%**  A literal .%. character |
| 33 | **%x**  x, for any.x. not listed above |

mysql> SELECT DATE\_FORMAT('1997-10-04 22:23:00', '%W %M %Y');

+---------------------------------------------------------+

| DATE\_FORMAT('1997-10-04 22:23:00', '%W %M %Y') |

+---------------------------------------------------------+

| Saturday October 1997 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

mysql> SELECT DATE\_FORMAT('1997-10-04 22:23:00'

-> '%H %k %I %r %T %S %w');

+---------------------------------------------------------+

| DATE\_FORMAT('1997-10-04 22:23:00....... |

+---------------------------------------------------------+

| 22 22 10 10:23:00 PM 22:23:00 00 6 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## DATE\_SUB(date,INTERVAL expr unit)

This is similar to the DATE\_ADD() function.

## DAY(date)

The DAY() is a synonym for the DAYOFMONTH() function.

## DAYNAME(date)

Returns the name of the weekday for date.

mysql> SELECT DAYNAME('1998-02-05');

+---------------------------------------------------------+

| DAYNAME('1998-02-05') |

+---------------------------------------------------------+

| Thursday |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## DAYOFMONTH(date)

Returns the day of the month for date, in the range 0 to 31.

mysql> SELECT DAYOFMONTH('1998-02-03');

+---------------------------------------------------------+

| DAYOFMONTH('1998-02-03') |

+---------------------------------------------------------+

| 3 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## DAYOFWEEK(date)

Returns the weekday index for date (1 = Sunday, 2 = Monday, ., 7 = Saturday). These index values correspond to the ODBC standard.

mysql> SELECT DAYOFWEEK('1998-02-03');

+---------------------------------------------------------+

|DAYOFWEEK('1998-02-03') |

+---------------------------------------------------------+

| 3 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## DAYOFYEAR(date)

Returns the day of the year for date, in the range 1 to 366.

mysql> SELECT DAYOFYEAR('1998-02-03');

+---------------------------------------------------------+

| DAYOFYEAR('1998-02-03') |

+---------------------------------------------------------+

| 34 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## EXTRACT(unit FROM date)

The EXTRACT() function uses the same kinds of unit specifiers as DATE\_ADD() or DATE\_SUB(), but extracts parts from the date rather than performing date arithmetic.

mysql> SELECT EXTRACT(YEAR FROM '1999-07-02');

+---------------------------------------------------------+

| EXTRACT(YEAR FROM '1999-07-02') |

+---------------------------------------------------------+

| 1999 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

mysql> SELECT EXTRACT(YEAR\_MONTH FROM '1999-07-02 01:02:03');

+---------------------------------------------------------+

| EXTRACT(YEAR\_MONTH FROM '1999-07-02 01:02:03') |

+---------------------------------------------------------+

| 199907 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## FROM\_DAYS(N)

Given a day number N, returns a DATE value.

mysql> SELECT FROM\_DAYS(729669);

+---------------------------------------------------------+

| FROM\_DAYS(729669) |

+---------------------------------------------------------+

| 1997-10-07 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

**Note** − Use FROM\_DAYS() with caution on old dates. It is not intended for use with values that precede the advent of the Gregorian calendar (1582).

## FROM\_UNIXTIME(unix\_timestamp)

## FROM\_UNIXTIME(unix\_timestamp,format)

Returns a representation of the **unix\_timestamp** argument as a value in 'YYYY-MM-DD HH:MM:SS or YYYYMMDDHHMMSS format, depending on whether the function is used in a string or in a numeric context. The value is expressed in the current time zone. The unix\_timestamp argument is an internal timestamp values, which are produced by the **UNIX\_TIMESTAMP()** function.

If the format is given, the result is formatted according to the format string, which is used in the same way as is listed in the entry for the **DATE\_FORMAT()** function.

mysql> SELECT FROM\_UNIXTIME(875996580);

+---------------------------------------------------------+

| FROM\_UNIXTIME(875996580) |

+---------------------------------------------------------+

| 1997-10-04 22:23:00 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## HOUR(time)

Returns the hour for time. The range of the return value is 0 to 23 for time-of-day values. However, the range of TIME values actually is much larger, so HOUR can return values greater than 23.

mysql> SELECT HOUR('10:05:03');

+---------------------------------------------------------+

| HOUR('10:05:03') |

+---------------------------------------------------------+

| 10 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## LAST\_DAY(date)

Takes a date or datetime value and returns the corresponding value for the last day of the month. Returns NULL if the argument is invalid.

mysql> SELECT LAST\_DAY('2003-02-05');

+---------------------------------------------------------+

| LAST\_DAY('2003-02-05') |

+---------------------------------------------------------+

| 2003-02-28 |

+---------------------------------------------------------+

1 row in set (0.00 sec)

## LOCALTIME and LOCALTIME()

LOCALTIME and LOCALTIME() are synonyms for NOW().

## LOCALTIMESTAMP and LOCALTIMESTAMP()

LOCALTIMESTAMP and LOCALTIMESTAMP() are synonyms for NOW().

## Types of SQL Aggregation Function

### **COUNT FUNCTION**

* COUNT function is used to Count the number of rows in a database table. It can work on both numeric and non-numeric data types.
* COUNT function uses the COUNT(\*) that returns the count of all the rows in a specified table. COUNT(\*) considers duplicate and Null.

**Syntax**

1. COUNT(\*)
2. or
3. COUNT( [ALL|DISTINCT] expression )

**Sample table:**

**PRODUCT\_MAST**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **PRODUCT** | **COMPANY** | **QTY** | **RATE** | **COST** |
| Item1 | Com1 | 2 | 10 | 20 |
| Item2 | Com2 | 3 | 25 | 75 |
| Item3 | Com1 | 2 | 30 | 60 |
| Item4 | Com3 | 5 | 10 | 50 |
| Item5 | Com2 | 2 | 20 | 40 |
| Item6 | Cpm1 | 3 | 25 | 75 |
| Item7 | Com1 | 5 | 30 | 150 |
| Item8 | Com1 | 3 | 10 | 30 |
| Item9 | Com2 | 2 | 25 | 50 |
| Item10 | Com3 | 4 | 30 | 120 |

**Example: COUNT()**

1. SELECT COUNT(\*)
2. FROM PRODUCT\_MAST;

**Output:**

**10**

**Example: COUNT with WHERE**

1. SELECT COUNT(\*)
2. FROM PRODUCT\_MAST;
3. WHERE RATE>=20;

**Output:**

7

**Example: COUNT() with DISTINCT**

1. SELECT COUNT(DISTINCT COMPANY)
2. FROM PRODUCT\_MAST;

**Output:**

3

**Example: COUNT() with GROUP BY**

1. SELECT COMPANY, COUNT(\*)
2. FROM PRODUCT\_MAST
3. GROUP BY COMPANY;

**Output:**

Com1 5

Com2 3

Com3 2

**Example: COUNT() with HAVING**

1. SELECT COMPANY, COUNT(\*)
2. FROM PRODUCT\_MAST
3. GROUP BY COMPANY
4. HAVING COUNT(\*)>2;

**Output:**

Com1 5

Com2 3

### **2. SUM Function**

Sum function is used to calculate the sum of all selected columns. It works on numeric fields only.

**Syntax**

1. SUM()
2. or
3. SUM( [ALL|DISTINCT] expression )

**Example: SUM()**

1. SELECT SUM(COST)
2. FROM PRODUCT\_MAST;

670

**Example: SUM() with WHERE**

1. SELECT SUM(COST)
2. FROM PRODUCT\_MAST
3. WHERE QTY>3;

**Output:**

320

**Example: SUM() with GROUP BY**

1. SELECT SUM(COST)
2. FROM PRODUCT\_MAST
3. WHERE QTY>3
4. GROUP BY COMPANY;

**Output:**

Com1 150

Com2 170

**Example: SUM() with HAVING**

1. SELECT COMPANY, SUM(COST)
2. FROM PRODUCT\_MAST
3. GROUP BY COMPANY
4. HAVING SUM(COST)>=170;

**Output:**

Com1 335

Com3 170

### **3. AVG function**

The AVG function is used to calculate the average value of the numeric type. AVG function returns the average of all non-Null values.

**Syntax**

1. AVG()
2. or
3. AVG( [ALL|DISTINCT] expression )

**Example:**

1. SELECT AVG(COST)
2. FROM PRODUCT\_MAST;

**Output:**

67.00

### **4. MAX Function**

MAX function is used to find the maximum value of a certain column. This function determines the largest value of all selected values of a column.

**Syntax**

1. MAX()
2. or
3. MAX( [ALL|DISTINCT] expression )

**Example:**

1. SELECT MAX(RATE)
2. FROM PRODUCT\_MAST;

30

### **5. MIN Function**

MIN function is used to find the minimum value of a certain column. This function determines the smallest value of all selected values of a column.

**Syntax**

1. MIN()
2. or
3. MIN( [ALL|DISTINCT] expression )

**Example:**

1. SELECT MIN(RATE)
2. FROM PRODUCT\_MAST;

**Output:**

10

* 1. **List each country name where the population is larger than that of 'Russia'.**

world(name, continent, area, population, gdp)

WORLD

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **name** | **continent** | **area** | **population** | **gdp** |
| Afghanistan | Asia | 652230 | 25500100 | 20343000000 |
| Albania | Europe | 28748 | 2831741 | 12960000000 |
| Algeria | Africa | 2381741 | 37100000 | 188681000000 |
| Andorra | Europe | 468 | 78115 | 3712000000 |
| Angola | Africa | 1246700 | 20609294 | 100990000000 |

SELECT name FROM world WHERE continent =

(SELECT continent

FROM world WHERE name = 'Brazil')

### 2) World Country Profile: Aggregate functions

This tutorial is about aggregate functions such as COUNT, SUM and AVG. An aggregate function takes many values and delivers just one value. For example the function SUM would aggregate the values 2, 4 and 5 to deliver the single value 11.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **name** | **continent** | **area** | **population** | **gdp** |
| Afghanistan | Asia | 652230 | 25500100 | 20343000000 |
| Albania | Europe | 28748 | 2831741 | 12960000000 |
| Algeria | Africa | 2381741 | 37100000 | 188681000000 |
| Andorra | Europe | 468 | 78115 | 3712000000 |
| Angola | Africa | 1246700 | 20609294 | 100990000000 |
| ... | | | | |

1. Show the total **population** of the world.

world(**name**, **continent**, **area**, **population**, **gdp**)

Select SUM(population) from world;

* 1. **List all the continents - just once each.**
  2. Give the total GDP of Africa

Top of Form

* 1. How many countries have an **area** of at least 1000000
  2. What is the total population of ('Estonia', 'Latvia', 'Lithuania')

3) Select SUM(gdp) from world where continent = 'Africa';

2) Select Distinct continent from World;

4) Select count(name) from World where area >=1000000;

**TRIGGER:**

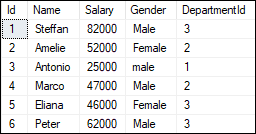
1. **CREATE** **TABLE** Employee
2. (
3. Id **INT** **PRIMARY** **KEY**,
4. **Name** **VARCHAR**(45),
5. Salary **INT**,
6. Gender **VARCHAR**(12),
7. DepartmentId **INT**
8. )

Next, we will insert some record into this table as follows:

1. **INSERT** **INTO** Employee **VALUES** (1,'Steffan', 82000, 'Male', 3),
2. (2,'Amelie', 52000, 'Female', 2),
3. (3,'Antonio', 25000, 'male', 1),
4. (4,'Marco', 47000, 'Male', 2),
5. (5,'Eliana', 46000, 'Female', 3)

We can verify the insert operation by using the SELECT statement. We will get the below output:

1. **SELECT** \* **FROM** Employee;



We will also create another table named '**Employee\_Audit\_Test'** to automatically store transaction records of each operation, such as INSERT, UPDATE, or DELETE on the Employee table:

1. **CREATE** **TABLE** Employee\_Audit\_Test
2. (
3. Id **int** IDENTITY,
4. Audit\_Action text
5. )

Now, we will **create a trigger that stores transaction records of each insert operation** on the Employee table into the Employee\_Audit\_Test table. Here we are going to create the insert trigger using the below statement:

1. **CREATE** **TRIGGER** trInsertEmployee
2. **ON** Employee
3. **FOR** **INSERT**
4. **AS**
5. **BEGIN**
6. **Declare** @Id **int**
7. **SELECT** @Id = Id **from** inserted
8. **INSERT** **INTO** Employee\_Audit\_Test
9. **VALUES** ('New employee with Id = ' + CAST(@Id **AS** **VARCHAR**(10)) + ' is added at ' + CAST(Getdate() **AS** **VARCHAR**(22)))
10. **END**

After creating a trigger, we will try to add the following record into the table:

1. **INSERT** **INTO** Employee **VALUES** (6,'Peter', 62000, 'Male', 3)

If no error is found, execute the SELECT statement to check the audit records. We will get the output as follows:

Triggers in SQL Server

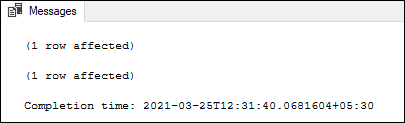
We are going to **create another trigger to store transaction records of each delete operation** on the Employee table into the Employee\_Audit\_Test table. We can create the delete trigger using the below statement:

1. **CREATE** **TRIGGER** trDeleteEmployee
2. **ON** Employee
3. **FOR** **DELETE**
4. **AS**
5. **BEGIN**
6. **Declare** @Id **int**
7. **SELECT** @Id = Id **from** deleted
8. **INSERT** **INTO** Employee\_Audit\_Test
9. **VALUES** ('An existing employee with Id = ' + CAST(@Id **AS** **VARCHAR**(10)) + ' is deleted at ' + CAST(Getdate() **AS** **VARCHAR**(22)))
10. **END**

After creating a trigger, we will delete a record from the Employee table:

1. **DELETE** **FROM** Employee **WHERE** Id = 2;

If no error is found, it gives the message as below:



Finally, execute the SELECT statement to check the audit records:

Triggers in SQL Server

In both the triggers code, you will notice these lines:

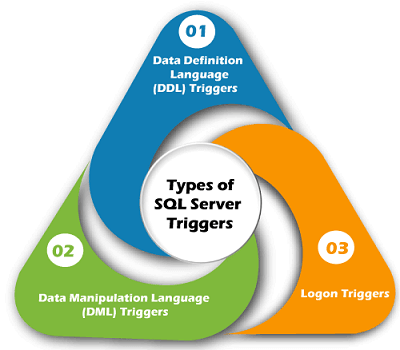
1. **SELECT** @Id = Id **from** inserted
2. **SELECT** @Id = Id **from** deleted

Here inserted and deleted are special tables used by the SQL Server. The inserted table keeps the copy of the row when you insert a new row into the actual table. And the deleted table keeps the copy of the row you have just deleted from the actual table.

## Types of SQL Server Triggers

We can categorize the triggers in SQL Server in mainly three types:

1. Data Definition Language (DDL) Triggers
2. Data Manipulation Language (DML) Triggers
3. Logon Triggers



**SQL pattern**

MySQL provides standard SQL pattern matching as well as a form of pattern matching based on extended regular expressions similar to those used by Unix utilities such as vi, grep, and sed.

SQL pattern matching allows you to search for patterns in data if you don't know the exact word or phrase you are seeking. This kind of [SQL](https://www.thoughtco.com/sql-fundamentals-1019780) query uses wildcard characters to match a pattern, rather than specifying it exactly. For example, you can use the wildcard "C%" to match any string beginning with a capital C.

To find names beginning with b:

mysql> SELECT \* FROM pet WHERE name LIKE 'b%';

+--------+--------+---------+------+------------+------------+

| name | owner | species | sex | birth | death |

+--------+--------+---------+------+------------+------------+

| Buffy | Harold | dog | f | 1989-05-13 | NULL |

| Bowser | Diane | dog | m | 1989-08-31 | 1995-07-29 |

+--------+--------+---------+------+------------+------------+

To find names ending with fy:

mysql> SELECT \* FROM pet WHERE name LIKE '%fy';

+--------+--------+---------+------+------------+-------+

| name | owner | species | sex | birth | death |

+--------+--------+---------+------+------------+-------+

| Fluffy | Harold | cat | f | 1993-02-04 | NULL |

| Buffy | Harold | dog | f | 1989-05-13 | NULL |

+--------+--------+---------+------+------------+-------+

To find names containing a w:

mysql> SELECT \* FROM pet WHERE name LIKE '%w%';

+----------+-------+---------+------+------------+------------+

| name | owner | species | sex | birth | death |

+----------+-------+---------+------+------------+------------+

| Claws | Gwen | cat | m | 1994-03-17 | NULL |

| Bowser | Diane | dog | m | 1989-08-31 | 1995-07-29 |

| Whistler | Gwen | bird | NULL | 1997-12-09 | NULL |

+----------+-------+---------+------+------------+------------+

To find names containing exactly five characters, use five instances of the \_ pattern character:

mysql> SELECT \* FROM pet WHERE name LIKE '\_\_\_\_\_';

+-------+--------+---------+------+------------+-------+

| name | owner | species | sex | birth | death |

+-------+--------+---------+------+------------+-------+

| Claws | Gwen | cat | m | 1994-03-17 | NULL |

| Buffy | Harold | dog | f | 1989-05-13 | NULL |

+-------+--------+---------+------+------------+-------+

## Using the LIKE Operator

To use a wildcard expression in an SQL query, use the LIKE operator in a WHERE clause, and enclose the pattern within single quotation marks.

## Using the % Wildcard to Perform a Simple Search

To search for any employee in your database with a last name beginning with the letter C, use the following Transact-SQL statement:

02:43

**How to Find the Median in a Set of Data**

SELECT \*  
 FROM employees  
 WHERE last\_name LIKE 'C%'

## Omitting Patterns Using the NOT Keyword

Use the NOT keyword to select records that don't match the pattern. For example, this query returns all records whose name last does not begin with C:

SELECT \*  
 FROM employees  
 WHERE last\_name NOT LIKE 'C%'

## Matching a Pattern Anywhere Using the % Wildcard Twice

Use two instances of the **%** wildcard to match a particular pattern anywhere. This example returns all records that contain a C anywhere in the last name:

SELECT \*  
 FROM employees  
 WHERE last\_name LIKE '%C%'

## Finding a Pattern Match at a Specific Position

Use the **\_** wildcard to return data at a specific location. This example matches only if C occurs at the third position of the last name column:

SELECT \*  
 FROM employees  
 WHERE last\_name LIKE '\_ \_C%'

Supported Wildcard Expressions in Transact SQL

There are several wildcard expressions supported by Transact SQL:

* The **%** wildcard matches zero or more characters of any type and can be used to define wildcards both before and after the pattern. If you're familiar with DOS pattern matching, it's the equivalent of the \* wildcard in that syntax.
* The **\_** wildcard matches exactly one character of any type. It's the equivalent of the **?** wildcard in DOS pattern matching.
* Specify a list of characters by enclosing them in square brackets. For example, the wildcard **[aeiou]** matches any vowel.
* Specify a range of characters by enclosing the range in square brackets. For example, the wildcard **[a-m]** matches any letter in the first half of the alphabet.
* Negate a range of characters by including the carat character immediately inside of the opening square bracket. For example, **[^aeiou]** matches any non-vowel character while **[^a-m]** matches any character not in the first half of the alphabet.

Combining Wildcards for Complex Patterns

Combine these wildcards in complex patterns to perform more advanced queries. For example, suppose you need to construct a list of all of your employees who have names that begin with a letter from the first half of the alphabet but do *not*end with a vowel. You could use the following query:

SELECT \*  
 FROM employees  
 WHERE last\_name LIKE '[a-m]%[^aeiou]'

Similarly, you could construct a list of all employees with last names consisting of exactly four characters by using four instances of the **\_** pattern:

SELECT \*  
 FROM employees  
 WHERE last\_name LIKE '\_\_\_\_'

As you can tell, the use of SQL pattern matching capabilities offers database users the ability to go beyond simple text queries and perform advanced searching operations.

* 1. Write the following inserts, deletes or updates in SQL, using the university

schema.

a. Increase the salary of each instructor in the Comp. Sci. department

by 10%.

b. Delete all courses that have never been offered (that is, do not occur

in the section relation).

10 Chapter 3 Introduction to SQL

c. Insert every student whose tot cred attribute is greater than 100 as an

instructor in the same department, with a salary of $10,000.

Answers:

**a. Increase the salary of each instructor in the Comp. Sci. department by 10%.**

update instructor

set salary = salary \* 1.10

where dept name = ’Comp. Sci.’

**b. Delete all courses that have never been offered (that is, do not occur in the section relation).**

delete from course

where course id not in

(select course id from section)

c**. Insert every student whose tot cred attribute is greater than 100 as an instructor in the same department, with a salary of $10,000.**

insert into instructor

select ID, name, dept name, 10000

from student

where tot cred > 100

* 1. Consider the insurance database of Figure ??, where the primary keys

are underlined. Construct the following SQL queries for this relational database.

a. Find the total number of people who owned cars that were involved in accidents in 1989.

b. Add a new accident to the database; assume any values for required attributes.

c. Delete the Mazda belonging to “John Smith”

**a. Find the total number of people who owned cars that were involved in accidents in 1989.**

Note: this is not the same as the total number of accidents in 1989.

We must count people with several accidents only once.

select count (distinct name)

from accident, participated, person

where accident.report number = participated.report number

and participated.driver id = person.driver id

and date between date ’1989-00-00’ and date ’1989-12-31’

person (driver id, name, address)

car (license, model, year)

accident (report number, date, location)

owns (driver id, license)

participated (driver id, car, report number, damage amount)

**Figure ??. Insurance database.**

**b. Add a new accident to the database; assume any values for required attributes.**

We assume the driver was “Jones,” although it could be someone

else. Also, we assume “Jones” owns one Toyota. First we must find

the license of the given car. Then the participated and accidentrelations

must be updated in order to both record the accident and tie it to the

given car. We assume values “Berkeley” for location, ’2001-09-01’ for

date and date, 4007 for report number and 3000 for damage amount.

insert into accident

values (4007, ’2001-09-01’, ’Berkeley’)

insert into participated

select o.driver id, c.license, 4007, 3000

from person p, owns o, car c

where p.name = ’Jones’ and p.driver id = o.driver id and

o.license = c.license and c.model = ’Toyota’

**c. Delete the Mazda belonging to “John Smith”.**

Since model is not a key of the car relation, we can either assume

that only one of John Smith’s cars is a Mazda, or delete all of John

Smith’s Mazdas (the query is the same). Again assume name is a key

for person.

delete car

where model = ’Mazda’ and license in

(select license

from person p, owns o

where p.name = ’John Smith’ and p.driver id = o.driver id)

Note: The owns, accident and participated records associated with the

Mazda still exist.

* 1. The SQL like operator is case sensitive, but the lower() function on strings can be used to perform case insensitive matching. To show how, write a query that finds departments whose names contain the string “sci” as a substring, regardless of the case

select dept\_name

from department

where lower(dept name) like ’%sci%’

* 1. Consider the relational database of Figure Give an expression in SQL for

each of the following queries.

a. Modify the database so that Jones now lives in Newtown.

b. Give all managers of First Bank Corporation a 10 percent raise unless the salary becomes greater than $100,000; in such cases, give only a 3 percent raise

* 1. update employee

set city = ’Newton’

where person name = ’Jones

* 1. update works T

set T.salary = T.salary \* 1.03

where T.employee name in (select manager name

from manages)

and T.salary \* 1.1 > 100000

and T.company name = ’First Bank Corporation’

update works T

set T.salary = T.salary \* 1.1

where T.employee name in (select manager name

from manages)

and T.salary \* 1.1 <= 100000

and T.company name = ’First Bank Corporation’

The above updates would give different results if executed in the

opposite order. We give below a safer solution using the case statement.

update works T

set T.salary = T.salary ∗

(case

when (T.salary ∗ 1.1 > 100000) then 1.03

else 1.1

)

where T.employee name in (select manager name

from manages) and

T.company name = ’First Bank Corporation