**SQL**

**Various *Syntax* in SQL**

## Operator in SQL

SQL Arithmetic Operators

SQL Comparison Operators

SQL Logical Operators

# SQL - Expressions

## Boolean Expressions

## Numeric Expression

## Date Expressions

## SQL - CREATE Database

## SQL - DROP or DELETE Database

## SQL - SELECT Database, USE Statement

## SQL - CREATE Table

## SQL - DROP or DELETE Table

## SQL - INSERT Query

## SQL - SELECT Query

## SQL - WHERE Clause

## SQL - AND and OR Conjunctive Operators

## SQL - UPDATE Query

## SQL - DELETE Query

## SQL - LIKE Clause

## SQL - TOP, LIMIT or ROWNUM Clause

## SQL - ORDER BY Clause

## SQL - Group By

## SQL - Distinct Keyword

## SQL - SORTING Results

## SQL – Constraints

## SQL - NOT NULL Constraint

## SQL - DEFAULT Constraint

## SQL - UNIQUE Constraint

## SQL - Primary Key

## SQL - Foreign Key

# SQL - CHECK Constraint

SQL - INDEX Constraint

# SQL - Using Joins

# SQL - INNER JOINS

# SQL - LEFT JOINS

# SQL - RIGHT JOINS

# SQL - FULL JOINS

# SQL - SELF JOINS

# SQL - CARTESIAN or CROSS JOINS

# SQL - UNIONS CLAUSE

# SQL - INTERSECT Clause

# SQL - EXCEPT Clause

SQL - NULL Values

SQL - Alias ***Syntax***

SQL – Indexes

SQL - ALTER TABLE Command

SQL - TRUNCATE TABLE Command

# SQL - Using Views

# SQL - Having Clause

# SQL - Transactions

# SQL - Wildcard Operators

# SQL - Date Functions

# SQL - Temporary Tables

# SQL - Clone Tables

# SQL - Sub Queries

# SQL - Using Sequences

# SQL - Handling Duplicates

# SQL - Injection

# SQL Interview Questions

Various ***Syntax*** in SQL

All the ***Example***s given in this tutorial have been tested with a MySQL server.

SQL SELECT Statement

|  |
| --- |
| SELECT column1, column2....columnN  FROM table\_name; |

SQL DISTINCT Clause

|  |
| --- |
| SELECT DISTINCT column1, column2....columnN  FROM table\_name; |

SQL WHERE Clause

|  |
| --- |
| SELECT column1, column2....columnN  FROM table\_name  WHERE CONDITION; |

SQL AND/OR Clause

|  |
| --- |
| SELECT column1, column2....columnN  FROM table\_name  WHERE CONDITION-1 {AND|OR} CONDITION-2; |

SQL IN Clause

|  |
| --- |
| SELECT column1, column2....columnN  FROM table\_name  WHERE column\_name IN (val-1, val-2,...val-N); |

SQL BETWEEN Clause

|  |
| --- |
| SELECT column1, column2....columnN  FROM table\_name  WHERE column\_name BETWEEN val-1 AND val-2; |

SQL LIKE Clause

|  |
| --- |
| SELECT column1, column2....columnN  FROM table\_name  WHERE column\_name LIKE { PATTERN }; |

SQL ORDER BY Clause

|  |
| --- |
| SELECT column1, column2....columnN  FROM table\_name  WHERE CONDITION  ORDER BY column\_name {ASC|DESC}; |

SQL GROUP BY Clause

|  |
| --- |
| SELECT SUM(column\_name)  FROM table\_name  WHERE CONDITION  GROUP BY column\_name; |

SQL COUNT Clause

|  |
| --- |
| SELECT COUNT(column\_name)  FROM table\_name  WHERE CONDITION; |

SQL HAVING Clause

|  |
| --- |
| SELECT SUM(column\_name)  FROM table\_name  WHERE CONDITION  GROUP BY column\_name  HAVING (arithematic function condition); |

SQL CREATE TABLE Statement

|  |
| --- |
| CREATE TABLE table\_name(  column1 datatype,  column2 datatype,  column3 datatype,  .....  columnN datatype,  PRIMARY KEY( one or more columns )  ); |

SQL DROP TABLE Statement

|  |
| --- |
| DROP TABLE table\_name; |

SQL CREATE INDEX Statement

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

SQL DROP INDEX Statement

|  |
| --- |
| ALTER TABLE table\_name  DROP INDEX index\_name; |

SQL DESC Statement

|  |
| --- |
| DESC table\_name; |

SQL TRUNCATE TABLE Statement

|  |
| --- |
| TRUNCATE TABLE table\_name; |

SQL ALTER TABLE Statement

|  |
| --- |
| ALTER TABLE table\_name {ADD|DROP|MODIFY} column\_name {data\_ype}; |

SQL ALTER TABLE Statement (Rename)

|  |
| --- |
| ALTER TABLE table\_name RENAME TO new\_table\_name; |

SQL INSERT INTO Statement

|  |
| --- |
| INSERT INTO table\_name( column1, column2....columnN)  VALUES ( value1, value2....valueN); |

SQL UPDATE Statement

|  |
| --- |
| UPDATE table\_name  SET column1 = value1, column2 = value2....columnN=valueN  [ WHERE CONDITION ]; |

SQL DELETE Statement

|  |
| --- |
| DELETE FROM table\_name  WHERE {CONDITION}; |

SQL CREATE DATABASE Statement

|  |
| --- |
| CREATE DATABASE database\_name; |

SQL DROP DATABASE Statement

|  |
| --- |
| DROP DATABASE database\_name; |

SQL USE Statement

|  |
| --- |
| USE database\_name; |

SQL COMMIT Statement

|  |
| --- |
| COMMIT; |

SQL ROLLBACK Statement

|  |
| --- |
| ROLLBACK; |

## Operator in SQL

* Arithmetic operators
* Comparison operators
* Logical operators
* Operators used to negate conditions

SQL Arithmetic Operators

Assume **'variable a'** holds 10 and **'variable b'** holds 20, then −

[Show ***Example***s](https://www.tutorialspoint.com/sql/sql-arithmetic-operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | ***Example*** |
| + (Addition) | Adds values on either side of the operator. | a + b will give 30 |
| - (Subtraction) | Subtracts right hand operand from left hand operand. | a - b will give -10 |
| \* (Multiplication) | Multiplies values on either side of the operator. | a \* b will give 200 |
| / (Division) | Divides left hand operand by right hand operand. | b / a will give 2 |
| % (Modulus) | Divides left hand operand by right hand operand and returns remainder. | b % a will give 0 |

SQL Comparison Operators

Assume **'variable a'** holds 10 and **'variable b'** holds 20, then −

[Show ***Example***s](https://www.tutorialspoint.com/sql/sql-comparison-operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | ***Example*** |
| = | Checks if the values of two operands are equal or not, if yes then condition becomes true. | (a = b) is not true. |
| != | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (a != b) is true. |
| <> | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (a <> b) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (a > b) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (a < b) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (a >= b) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (a <= b) is true. |
| !< | Checks if the value of left operand is not less than the value of right operand, if yes then condition becomes true. | (a !< b) is false. |
| !> | Checks if the value of left operand is not greater than the value of right operand, if yes then condition becomes true. | (a !> b) is true. |

SQL Logical Operators

Here is a list of all the logical operators available in SQL.

[Show ***Example***s](https://www.tutorialspoint.com/sql/sql-logical-operators.htm)

|  |  |
| --- | --- |
| **Sr.No.** | **Operator & Description** |
| 1 | **ALL**  The ALL operator is used to compare a value to all values in another value set. |
| 2 | **AND**  The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause. |
| 3 | **ANY**  The ANY operator is used to compare a value to any applicable value in the list as per the condition. |
| 4 | **BETWEEN**  The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value. |
| 5 | **EXISTS**  The EXISTS operator is used to search for the presence of a row in a specified table that meets a certain criterion. |
| 6 | **IN**  The IN operator is used to compare a value to a list of literal values that have been specified. |
| 7 | **LIKE**  The LIKE operator is used to compare a value to similar values using wildcard operators. |
| 8 | **NOT**  The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. **This is a negate operator.** |
| 9 | **OR**  The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause. |
| 10 | **IS NULL**  The NULL operator is used to compare a value with a NULL value. |
| 11 | **UNIQUE**  The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates). |

# SQL - Expressions

***Syntax***

Consider the basic ***Syntax*** of the SELECT statement as follows −

|  |
| --- |
| SELECT column1, column2, columnN  FROM table\_name  WHERE [CONDITION|EXPRESSION]; |

There are different types of SQL expressions, which are mentioned below −

* Boolean
* Numeric
* Date

## Boolean Expressions

SQL Boolean Expressions fetch the data based on matching a single value.

***Syntax***

|  |
| --- |
| SELECT column1, column2, columnN  FROM table\_name  WHERE SINGLE VALUE MATCHING EXPRESSION; |

CUSTOMERS table

SQL> SELECT \* FROM CUSTOMERS;

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

| 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 |

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

7 rows in set (0.00 sec)

***Example***

SQL> SELECT \* FROM CUSTOMERS WHERE SALARY = 10000;

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 7 | Muffy | 24 | Indore | 10000.00 |

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

1 row in set (0.00 sec)

## Numeric Expression

These expressions are used to perform any mathematical operation in any query.

***Syntax***

|  |
| --- |
| SELECT numerical\_expression as OPERATION\_NAME  [FROM table\_name  WHERE CONDITION] ; |

***Example***

SQL> SELECT (15 + 6) AS ADDITION

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

| ADDITION |

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

| 21 |

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

1 row in set (0.00 sec)

built-in functions like avg(), sum(), count(), etc., aggregate data calculations

SQL> SELECT COUNT(\*) AS "RECORDS" FROM CUSTOMERS;

+---------+

| RECORDS |

+---------+

| 7 |

+---------+

1 row in set (0.00 sec)

## Date Expressions

Date Expressions return current system date and time values −

SQL> SELECT CURRENT\_TIMESTAMP;

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

| Current\_Timestamp |

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

| 2009-11-12 06:40:23 |

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

1 row in set (0.00 sec)

Another date expression

SQL> SELECT GETDATE();;

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

| GETDATE |

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

| 2009-10-22 12:07:18.140 |

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

1 row in set (0.00 sec)

# SQL - CREATE Database

***Syntax***

|  |
| --- |
| CREATE DATABASE DatabaseName; |

Always the database name should be unique within the RDBMS.

***Example***

If you want to create a new database <testDB>

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

|  |
| --- |
| SQL> SHOW DATABASES; |

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

| Database |

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

| information\_schema |

| AMROOD |

| TUTORIALSPOINT |

| mysql |

| orig |

| test |

| testDB |

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

7 rows in set (0.00 sec)

# SQL - DROP or DELETE Database

## *Syntax*

|  |
| --- |
| DROP DATABASE DatabaseName; |

***Example***

to delete an existing database <testDB>, use the DROP DATABASE statement

SQL> DROP DATABASE testDB;

SQL> SHOW DATABASES;

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

| Database |

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

| information\_schema |

| AMROOD |

| TUTORIALSPOINT |

| mysql |

| orig |

| test |

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

6 rows in set (0.00 sec)

# SQL - SELECT Database, USE Statement

## *Syntax*

|  |
| --- |
| USE DatabaseName; |

***Example***

SQL> SHOW DATABASES;

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

| Database |

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

| information\_schema |

| AMROOD |

| TUTORIALSPOINT |

| mysql |

| orig |

| test |

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

6 rows in set (0.00 sec)

If you want to work with the AMROOD database

SQL> USE AMROOD;

# SQL - CREATE Table

## *Syntax*

|  |
| --- |
| CREATE TABLE table\_name(  column1 datatype,  column2 datatype,  column3 datatype,  .....  columnN datatype,  PRIMARY KEY( one or more columns )  ); |

## *Example*

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)

);

DESC command

***Syntax***

|  |
| --- |
| 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)

# SQL - DROP or DELETE Table

## *Syntax*

|  |
| --- |
| DROP TABLE table\_name; |

## *Example*

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)

SQL> DROP TABLE CUSTOMERS;

Query OK, 0 rows affected (0.01 sec)

SQL> DESC CUSTOMERS;

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

# SQL - INSERT Query

There are two basic ***Syntax***es of the INSERT INTO statement

***Syntax***

|  |
| --- |
| INSERT INTO TABLE\_NAME (column1, column2, column3,...columnN)  VALUES (value1, value2, value3,...valueN); |

***Syntax*** SQL INSERT INTO

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

### *Example*

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 );

INSERT INTO CUSTOMERS

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

CUSTOMERS table

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

| 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.

***Syntax***

|  |
| --- |
| INSERT INTO first\_table\_name [(column1, column2, ... columnN)]  SELECT column1, column2, ...columnN  FROM second\_table\_name  [WHERE condition]; |

# SQL - SELECT Query

## *Syntax*

|  |
| --- |
| SELECT column1, column2, columnN FROM table\_name; |

***Syntax***

SELECT \* FROM table\_name;

## *Example*

CUSTOMERS table

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

| 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 |

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

To fetch the ID, Name and Salary fields of the customers available in CUSTOMERS table.

SQL> SELECT ID, NAME, SALARY FROM CUSTOMERS;

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 |

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

To fetch all the fields of the CUSTOMERS table

SQL> SELECT \* FROM CUSTOMERS;

result

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

| 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 |

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

# SQL - WHERE Clause

## *Syntax*

|  |
| --- |
| SELECT column1, column2, columnN  FROM table\_name  WHERE [condition] |

***Note:***specify a condition using the [comparison or logical operators](https://www.tutorialspoint.com/sql/sql-operators.htm) like >, <, =, **LIKE, NOT**, etc.

***Example***

CUSTOMERS table

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

| 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 |

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

To 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;

result

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

| ID | NAME | SALARY |

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

| 4 | Chaitali | 6500.00 |

| 5 | Hardik | 8500.00 |

| 6 | Komal | 4500.00 |

| 7 | Muffy | 10000.00 |

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

To fetch the ID, Name and Salary fields from the CUSTOMERS table for a customer with the name Hardik.

***Note*:** that all the strings should be given inside single quotes (''). Whereas, numeric values should be given without any quote

SQL> SELECT ID, NAME, SALARY

FROM CUSTOMERS

WHERE NAME = 'Hardik';

result

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

| ID | NAME | SALARY |

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

| 5 | Hardik | 8500.00 |

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

# SQL - AND and OR Conjunctive Operators

## The AND Operator

The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause.

### *Syntax*

|  |
| --- |
| SELECT column1, column2, columnN  FROM table\_name  WHERE [condition1] AND [condition2]...AND [conditionN]; |

You can combine N number of conditions using the AND operator. For an action to be taken by the SQL statement, whether it be a transaction or a query, all conditions separated by the AND must be TRUE.

### *Example*

CUSTOMERS table

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

| 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 |

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

To fetch the ID, Name and Salary fields from the CUSTOMERS table, where the salary is greater than 2000 and the age is less than 25 years −

SQL> SELECT ID, NAME, SALARY

FROM CUSTOMERS

WHERE SALARY > 2000 AND age < 25;

result −

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

| ID | NAME | SALARY |

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

| 6 | Komal | 4500.00 |

| 7 | Muffy | 10000.00 |

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

## The OR Operator

The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.

### *Syntax*

|  |
| --- |
| SELECT column1, column2, columnN  FROM table\_name  WHERE [condition1] OR [condition2]...OR [conditionN] |

You can combine N number of conditions using the OR operator. For an action to be taken by the SQL statement, whether it be a transaction or query, the only any ONE of the conditions separated by the OR must be TRUE.

### *Example*

CUSTOMERS table

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

| 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 |

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

To fetch the ID, Name and Salary fields from the CUSTOMERS table, where the salary is greater than 2000 OR the age is less than 25 years.

SQL> SELECT ID, NAME, SALARY

FROM CUSTOMERS

WHERE SALARY > 2000 OR age < 25;

result −

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

| ID | NAME | SALARY |

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

| 3 | kaushik | 2000.00 |

| 4 | Chaitali | 6500.00 |

| 5 | Hardik | 8500.00 |

| 6 | Komal | 4500.00 |

| 7 | Muffy | 10000.00 |

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

# SQL - UPDATE Query

***Syntax***

|  |
| --- |
| UPDATE table\_name  SET column1 = value1, column2 = value2...., columnN = valueN  WHERE [condition]; |

You can combine N number of conditions using the AND or the OR operators.

***Example***

CUSTOMERS table

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

| 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 query will update the ADDRESS for a customer whose ID number is 6

SQL> UPDATE CUSTOMERS

SET ADDRESS = 'Pune'

WHERE ID = 6;

CUSTOMERS table

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

| 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 | Pune | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

If you want to modify all the ADDRESS and the SALARY column values in the CUSTOMERS table, you do not need to use the WHERE clause as the using UPDATE .

SQL> UPDATE CUSTOMERS

SET ADDRESS = 'Pune', SALARY = 1000.00;

CUSTOMERS table

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Pune | 1000.00 |

| 2 | Khilan | 25 | Pune | 1000.00 |

| 3 | kaushik | 23 | Pune | 1000.00 |

| 4 | Chaitali | 25 | Pune | 1000.00 |

| 5 | Hardik | 27 | Pune | 1000.00 |

| 6 | Komal | 22 | Pune | 1000.00 |

| 7 | Muffy | 24 | Pune | 1000.00 |

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

# SQL - DELETE Query

***Syntax***

|  |
| --- |
| DELETE FROM table\_name  WHERE [condition]; |

***Example***

CUSTOMERS table

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

| 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 query will DELETE a customer, whose ID is 6.

SQL> DELETE FROM CUSTOMERS

WHERE ID = 6;

CUSTOMERS table.

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

| 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 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

If you want to DELETE all the records from the CUSTOMERS table, you do not need to use the WHERE clause and the DELETE query −

SQL> DELETE FROM CUSTOMERS;

Now, the CUSTOMERS table would not have any record.

# SQL - LIKE Clause

The SQL **LIKE** clause is used to compare a value to similar values using wildcard operators. There are two wildcards used in conjunction with the LIKE operator.

* The percent sign (%)
* The underscore (\_)

The percent sign represents zero, one or multiple characters. The underscore represents a single number or character. These symbols can be used in combinations.

***Syntax***

|  |
| --- |
| SELECT FROM table\_name  WHERE column LIKE 'XXXX%' (or) |
| SELECT FROM table\_name  WHERE column LIKE '%XXXX%' (or) |
| SELECT FROM table\_name  WHERE column LIKE 'XXXX\_' (or) |
| SELECT FROM table\_name  WHERE column LIKE '\_XXXX' (or) |
| SELECT FROM table\_name  WHERE column LIKE '\_XXXX\_' |

Here, XXXX could be any numeric or string value.

***Example***

The WHERE part having different LIKE clause with '%' and '\_' operators

|  |  |
| --- | --- |
| **Sr.No.** | **Statement & Description** |
| 1 | **WHERE SALARY LIKE '200%'**  Finds any values that start with 200. |
| 2 | **WHERE SALARY LIKE '%200%'**  Finds any values that have 200 in any position. |
| 3 | **WHERE SALARY LIKE '\_00%'**  Finds any values that have 00 in the second and third positions. |
| 4 | **WHERE SALARY LIKE '2\_%\_%'**  Finds any values that start with 2 and are at least 3 characters in length. |
| 5 | **WHERE SALARY LIKE '%2'**  Finds any values that end with 2. |
| 6 | **WHERE SALARY LIKE '\_2%3'**  Finds any values that have a 2 in the second position and end with a 3. |
| 7 | **WHERE SALARY LIKE '2\_\_\_3'**  Finds any values in a five-digit number that start with 2 and end with 3. |

CUSTOMERS table.

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

| 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 |

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

***Example*** to display all the records from the CUSTOMERS table, where the SALARY starts with 200.

SQL> SELECT \* FROM CUSTOMERS

WHERE SALARY LIKE '200%';

result

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

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

# SQL - TOP, LIMIT or ROWNUM Clause

# *Syntax*

|  |
| --- |
| SELECT TOP number|percent column\_name(s)  FROM table\_name  WHERE [condition] |

***Example***

CUSTOMERS table

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

| 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 |

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

***Example*** to fetch the top 3 records from the CUSTOMERS table.

SQL> SELECT TOP 3 \* FROM CUSTOMERS;

result

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

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

***Example*** for MySQL server

SQL> SELECT \* FROM CUSTOMERS

LIMIT 3;

result

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

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

***Example*** Oracle server

SQL> SELECT \* FROM CUSTOMERS

WHERE ROWNUM <= 3;

result

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

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

# SQL - ORDER BY Clause

# *Syntax*

|  |
| --- |
| SELECT column-list  FROM table\_name  [WHERE condition]  [ORDER BY column1, column2, .. columnN] [ASC | DESC]; |

You can use more than one column in the ORDER BY clause.

***Example***

CUSTOMERS table

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

| 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 |

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

***Example*** to sort the result in an ascending order by the NAME and the SALARY

SQL> SELECT \* FROM CUSTOMERS

ORDER BY NAME, SALARY;

result

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

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

***Example*** to sort the result in the descending order by NAME.

SQL> SELECT \* FROM CUSTOMERS

ORDER BY NAME DESC;

result

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

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

# SQL - Group By

***Syntax***

|  |
| --- |
| SELECT column1, column2  FROM table\_name  WHERE [ conditions ]  GROUP BY column1, column2  ORDER BY column1, column2 |

The GROUP BY clause must follow the conditions in the WHERE clause and must precede the ORDER BY clause if one is used.

***Example***

CUSTOMERS table

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

| 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 |

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

to know the total amount of the salary on each customer,using GROUP BY query.

SQL> SELECT NAME, SUM(SALARY) FROM CUSTOMERS

GROUP BY NAME;

result

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

| NAME | SUM(SALARY) |

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

| Chaitali | 6500.00 |

| Hardik | 8500.00 |

| kaushik | 2000.00 |

| Khilan | 1500.00 |

| Komal | 4500.00 |

| Muffy | 10000.00 |

| Ramesh | 2000.00 |

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

CUSTOMERS table has the following records with duplicate names −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Ramesh | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | kaushik | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

To know the total amount of salary on each customer, using GROUP BY

SQL> SELECT NAME, SUM(SALARY) FROM CUSTOMERS

GROUP BY NAME;

result

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

| NAME | SUM(SALARY) |

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

| Hardik | 8500.00 |

| kaushik | 8500.00 |

| Komal | 4500.00 |

| Muffy | 10000.00 |

| Ramesh | 3500.00 |

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

# SQL - Distinct Keyword

***Syntax***

|  |
| --- |
| SELECT DISTINCT column1, column2,.....columnN  FROM table\_name  WHERE [condition] |

***Example***

CUSTOMERS table

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

| 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 |

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

First, let us see how the following SELECT query returns the duplicate salary records.

SQL> SELECT SALARY FROM CUSTOMERS

ORDER BY SALARY;

This would produce the following result, where the salary (2000) is coming twice which is a duplicate record from the original table.

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

| SALARY |

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

| 1500.00 |

| 2000.00 |

| 2000.00 |

| 4500.00 |

| 6500.00 |

| 8500.00 |

| 10000.00 |

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

let us use the DISTINCT keyword with the above SELECT query and see the result.

SQL> SELECT DISTINCT SALARY FROM CUSTOMERS

ORDER BY SALARY;

Result

here we do not have any duplicate entry.

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

| SALARY |

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

| 1500.00 |

| 2000.00 |

| 4500.00 |

| 6500.00 |

| 8500.00 |

| 10000.00 |

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

# SQL - SORTING Results

***Syntax***

|  |
| --- |
| SELECT column-list  FROM table\_name  [WHERE condition]  [ORDER BY column1, column2, .. columnN] [ASC | DESC]; |

***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 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

***Example*** to sort the result in an ascending order by NAME and SALARY.

SQL> SELECT \* FROM CUSTOMERS

ORDER BY NAME, SALARY;

result

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

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

***Example*** to sort the result in a descending order by NAME.

SQL> SELECT \* FROM CUSTOMERS

ORDER BY NAME DESC;

result

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

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

To fetch the rows with their own preferred order, used the SELECT query

SQL> SELECT \* FROM CUSTOMERS

ORDER BY (CASE ADDRESS

WHEN 'DELHI' THEN 1

WHEN 'BHOPAL' THEN 2

WHEN 'KOTA' THEN 3

WHEN 'AHMADABAD' THEN 4

WHEN 'MP' THEN 5

ELSE 100 END) ASC, ADDRESS DESC;

result

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

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

This will sort the customers by ADDRESS in your **ownoOrder** of preference first and in a natural order for the remaining addresses. Also, the remaining Addresses will be sorted in the reverse alphabetical order.

# SQL - Constraints

* [NOT NULL Constraint](https://www.tutorialspoint.com/sql/sql-not-null.htm) − Ensures that a column cannot have NULL value.
* [DEFAULT Constraint](https://www.tutorialspoint.com/sql/sql-default.htm) − Provides a default value for a column when none is specified.
* [UNIQUE Constraint](https://www.tutorialspoint.com/sql/sql-unique.htm) − Ensures that all values in a column are different.
* [PRIMARY Key](https://www.tutorialspoint.com/sql/sql-primary-key.htm) − Uniquely identifies each row/record in a database table.
* [FOREIGN Key](https://www.tutorialspoint.com/sql/sql-foreign-key.htm) − Uniquely identifies a row/record in any of the given database table.
* [CHECK Constraint](https://www.tutorialspoint.com/sql/sql-check.htm) − The CHECK constraint ensures that all the values in a column satisfies certain conditions.
* [INDEX](https://www.tutorialspoint.com/sql/sql-index.htm) − Used to create and retrieve data from the database very quickly.

Constraints can be specified when a table is created with the CREATE TABLE statement or you can use the ALTER TABLE statement to create constraints even after the table is created.

# SQL - NOT NULL Constraint

***Example***,creates a table CUSTOMERS and add five columns, ID NAME and AGE

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)

);

If CUSTOMERS table has already created, then add a NOT NULL constraint to the SALARY column in Oracle and MySQL

ALTER TABLE CUSTOMERS

MODIFY SALARY DECIMAL (18, 2) NOT NULL;

# SQL - DEFAULT Constraint

***Example***, creates a table CUSTOMERS.SALARY column is set to 5000.00 by default, so in case the INSERT INTO statement does not provide a value for this column, then by default this column would be set to 5000.00.

CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25) ,

SALARY DECIMAL (18, 2) DEFAULT 5000.00,

PRIMARY KEY (ID)

);

CUSTOMERS table has already created, add DEFAULT constraint to SALARY column

ALTER TABLE CUSTOMERS

MODIFY SALARY DECIMAL (18, 2) DEFAULT 5000.00;

Drop Default Constraint

***Syntax***,To drop a DEFAULT constraint

ALTER TABLE CUSTOMERS

ALTER COLUMN SALARY DROP DEFAULT;

# SQL - UNIQUE Constraint

***Example*** creates a table CUSTOMERS.AGE column is set to UNIQUE, so that you cannot have two records with the same age.

CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL UNIQUE,

ADDRESS CHAR (25) ,

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

CUSTOMERS table has already created, add a UNIQUE constraint to AGE column.

ALTER TABLE CUSTOMERS

MODIFY AGE INT NOT NULL UNIQUE;

***Syntax***, which supports naming the constraint in multiple columns as well.

ALTER TABLE CUSTOMERS

ADD CONSTRAINT myUniqueConstraint UNIQUE(AGE, SALARY);

DROP a UNIQUE Constraint

***Syntax*** To drop a UNIQUE constraint.

ALTER TABLE CUSTOMERS

DROP CONSTRAINT myUniqueConstraint;

***Syntax*** If you are using MySQL, then you can use the following −

|  |
| --- |
| ALTER TABLE CUSTOMERS  DROP INDEX myUniqueConstraint; |

# SQL - Primary Key

***Note*** − You would use these concepts while creating database tables.

Create Primary Key

***Syntax***

|  |
| --- |
| 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)); |

To create a PRIMARY KEY constraint on the "ID" column in the CUSTOMERS table already exists

|  |
| --- |
| ALTER TABLE CUSTOMER ADD PRIMARY KEY (ID); |

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

***Syntax***. For defining a PRIMARY KEY constraint on multiple columns,

|  |
| --- |
| 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, NAME)); |

To create a PRIMARY KEY constraint on the "ID" and "NAMES" columns when CUSTOMERS table already exists

***Syntax***.

ALTER TABLE CUSTOMERS

ADD CONSTRAINT PK\_CUSTID PRIMARY KEY (ID, NAME);

Delete Primary Key

You can clear the primary key constraints from the table with the

***Syntax***.

|  |
| --- |
| ALTER TABLE CUSTOMERS DROP PRIMARY KEY ; |

# SQL - Foreign Key

The relationship between 2 tables matches the Primary Key in one of the tables with a Foreign Key in the second table.

***Example***

CUSTOMERS table

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)

);

ORDERS table

CREATE TABLE ORDERS (

ID INT NOT NULL,

DATE DATETIME,

CUSTOMER\_ID INT references CUSTOMERS(ID),

AMOUNT double,

PRIMARY KEY (ID)

);

***Syntax***

|  |
| --- |
| ALTER TABLE ORDERS  ADD FOREIGN KEY (Customer\_ID) REFERENCES CUSTOMERS (ID); |

DROP a FOREIGN KEY Constraint

To drop a FOREIGN KEY constraint

ALTER TABLE ORDERS

DROP FOREIGN KEY;

# SQL - CHECK Constraint

***Example***

CUSTOMERS table, add a CHECK with AGE column, so that you cannot have any CUSTOMER who is below 18 years.

CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL CHECK (AGE >= 18),

ADDRESS CHAR (25) ,

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

CUSTOMERS table already created, add a CHECK constraint to AGE column

ALTER TABLE CUSTOMERS

MODIFY AGE INT NOT NULL CHECK (AGE >= 18 );

***Syntax***, which supports naming the constraint in multiple columns as well −

ALTER TABLE CUSTOMERS

ADD CONSTRAINT myCheckConstraint CHECK(AGE >= 18);

DROP a CHECK Constraint

***Syntax*** to drop a CHECK constraint..

|  |
| --- |
| ALTER TABLE CUSTOMERS  DROP CONSTRAINT myCheckConstraint; |

# SQL - INDEX Constraint

***Example***

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)

);

***Syntax*** to create an index on a single or multiple columns.

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

***Syntax*** to create an INDEX on the AGE column.

|  |
| --- |
| CREATE INDEX idx\_age  ON CUSTOMERS ( AGE ); |

DROP an INDEX Constraint

, ***Syntax*** To drop an INDEX constraint.

|  |
| --- |
| ALTER TABLE CUSTOMERS  DROP INDEX idx\_age; |

Dropping Constraints

Any constraint that you have defined can be dropped using the ALTER TABLE command with the DROP CONSTRAINT option.

***Example***, to drop the primary key constraint in the EMPLOYEES table

ALTER TABLE EMPLOYEES DROP CONSTRAINT EMPLOYEES\_PK;

***Example***, to drop the primary key constraint for a table in Oracle

ALTER TABLE EMPLOYEES DROP PRIMARY KEY;

## Integrity Constraints

**Referential Integrity (RI)**.

These constraints include Primary Key, Foreign Key, Unique Constraints and other constraints which are mentioned above.

# SQL - Using Joins

Table 1 − CUSTOMERS Table

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

| 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 |

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

Table 2 − ORDERS Table

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

|OID | DATE | CUSTOMER\_ID | AMOUNT |

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

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

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

Use SELECT statement

SQL> SELECT ID, NAME, AGE, AMOUNT

FROM CUSTOMERS, ORDERS

WHERE CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

result.

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

| ID | NAME | AGE | AMOUNT |

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

| 3 | kaushik | 23 | 3000 |

| 3 | kaushik | 23 | 1500 |

| 2 | Khilan | 25 | 1560 |

| 4 | Chaitali | 25 | 2060 |

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

***Note***: that the join is performed in the WHERE clause. Several operators can be used to join tables, such as =, <, >, <>, <=, >=, !=, BETWEEN, LIKE, and NOT; they can all be used to join tables.the most common operator is equal to symbol.

There are different types of joins available in SQL −

* [INNER JOIN](https://www.tutorialspoint.com/sql/sql-inner-joins.htm) − returns rows when there is a match in both tables.
* [LEFT JOIN](https://www.tutorialspoint.com/sql/sql-left-joins.htm) − returns all rows from the left table, even if there are no matches in the right table.
* [RIGHT JOIN](https://www.tutorialspoint.com/sql/sql-right-joins.htm) − returns all rows from the right table, even if there are no matches in the left table.
* [FULL JOIN](https://www.tutorialspoint.com/sql/sql-full-joins.htm) − returns rows when there is a match in one of the tables.
* [SELF JOIN](https://www.tutorialspoint.com/sql/sql-self-joins.htm) − is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.
* [CARTESIAN JOIN](https://www.tutorialspoint.com/sql/sql-cartesian-joins.htm) − returns the Cartesian product of the sets of records from the two or more joined tables.

# SQL - INNER JOINS

**INNER JOIN** Or **EQUIJOIN**.

***Syntax***

|  |
| --- |
| SELECT table1.column1, table2.column2...  FROM table1  INNER JOIN table2  ON table1.common\_field = table2.common\_field; |

***Example***

Table 1 − CUSTOMERS Table is as follows.

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

| 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 |

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

Table 2 − ORDERS Table is as follows.

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

| OID | DATE | CUSTOMER\_ID | AMOUNT |

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

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

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

use INNER JOIN

SQL> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

INNER JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

result.

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

| ID | NAME | AMOUNT | DATE |

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

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

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

# SQL - LEFT JOINS

***Syntax***

|  |
| --- |
| SELECT table1.column1, table2.column2...  FROM table1  LEFT JOIN table2  ON table1.common\_field = table2.common\_field; |

the given condition could be any given expression based on your requirement.

***Example***

Table 1 − CUSTOMERS Table is as follows.

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

| 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 |

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

Table 2 − Orders Table is as follows.

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

| OID | DATE | CUSTOMER\_ID | AMOUNT |

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

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

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

Use LEFT JOIN.

SQL> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

LEFT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

result −

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

| ID | NAME | AMOUNT | DATE |

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

| 1 | Ramesh | NULL | NULL |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

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

# SQL - RIGHT JOINS

***Syntax***

|  |
| --- |
| SELECT table1.column1, table2.column2...  FROM table1  RIGHT JOIN table2  ON table1.common\_field = table2.common\_field; |

***Example***

Table 1 − CUSTOMERS Table is as follows.

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

| 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 |

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

Table 2 − ORDERS Table is as follows.

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

|OID | DATE | CUSTOMER\_ID | AMOUNT |

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

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

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

Use RIGHT JOIN

SQL> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

RIGHT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

result

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

| ID | NAME | AMOUNT | DATE |

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

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

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

# SQL - FULL JOINS

***Syntax***

|  |
| --- |
| SELECT table1.column1, table2.column2...  FROM table1  FULL JOIN table2  ON table1.common\_field = table2.common\_field; |

***Example***

Table 1 − CUSTOMERS Table is as follows.

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

| 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 |

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

Table 2 − ORDERS Table is as follows.

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

|OID | DATE | CUSTOMER\_ID | AMOUNT |

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

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

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

Use FULL JOIN.

SQL> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

FULL JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

result

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

| ID | NAME | AMOUNT | DATE |

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

| 1 | Ramesh | NULL | NULL |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

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

If your Database does not support FULL JOIN (MySQL does not support FULL JOIN), then you can use **UNION ALL** clause to combine these two JOINS.

SQL> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

LEFT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID

UNION ALL

SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

RIGHT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID

# SQL - SELF JOINS

## *Syntax*

|  |
| --- |
| SELECT a.column\_name, b.column\_name...  FROM table1 a, table1 b  WHERE a.common\_field = b.common\_field; |

## *Example*

CUSTOMERS Table is as follows.

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

| 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 |

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

Use SELF JOIN

SQL> SELECT a.ID, b.NAME, a.SALARY

FROM CUSTOMERS a, CUSTOMERS b

WHERE a.SALARY < b.SALARY;

result

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

| ID | NAME | SALARY |

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

| 2 | Ramesh | 1500.00 |

| 2 | kaushik | 1500.00 |

| 1 | Chaitali | 2000.00 |

| 2 | Chaitali | 1500.00 |

| 3 | Chaitali | 2000.00 |

| 6 | Chaitali | 4500.00 |

| 1 | Hardik | 2000.00 |

| 2 | Hardik | 1500.00 |

| 3 | Hardik | 2000.00 |

| 4 | Hardik | 6500.00 |

| 6 | Hardik | 4500.00 |

| 1 | Komal | 2000.00 |

| 2 | Komal | 1500.00 |

| 3 | Komal | 2000.00 |

| 1 | Muffy | 2000.00 |

| 2 | Muffy | 1500.00 |

| 3 | Muffy | 2000.00 |

| 4 | Muffy | 6500.00 |

| 5 | Muffy | 8500.00 |

| 6 | Muffy | 4500.00 |

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

# SQL - CARTESIAN or CROSS JOINS

**CARTESIAN JOIN** or **CROSS JOIN**

***Syntax***

|  |
| --- |
| SELECT table1.column1, table2.column2...  FROM table1, table2 [, table3 ] |

***Example***

Table 1 − CUSTOMERS table is as follows.

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

| 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 |

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

Table 2: ORDERS Table is as follows −

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

|OID | DATE | CUSTOMER\_ID | AMOUNT |

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

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

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

Use CARTESIAN JOIN

SQL> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS, ORDERS;

result

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

| ID | NAME | AMOUNT | DATE |

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

| 1 | Ramesh | 3000 | 2009-10-08 00:00:00 |

| 1 | Ramesh | 1500 | 2009-10-08 00:00:00 |

| 1 | Ramesh | 1560 | 2009-11-20 00:00:00 |

| 1 | Ramesh | 2060 | 2008-05-20 00:00:00 |

| 2 | Khilan | 3000 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 2 | Khilan | 2060 | 2008-05-20 00:00:00 |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1560 | 2009-11-20 00:00:00 |

| 3 | kaushik | 2060 | 2008-05-20 00:00:00 |

| 4 | Chaitali | 3000 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | 3000 | 2009-10-08 00:00:00 |

| 5 | Hardik | 1500 | 2009-10-08 00:00:00 |

| 5 | Hardik | 1560 | 2009-11-20 00:00:00 |

| 5 | Hardik | 2060 | 2008-05-20 00:00:00 |

| 6 | Komal | 3000 | 2009-10-08 00:00:00 |

| 6 | Komal | 1500 | 2009-10-08 00:00:00 |

| 6 | Komal | 1560 | 2009-11-20 00:00:00 |

| 6 | Komal | 2060 | 2008-05-20 00:00:00 |

| 7 | Muffy | 3000 | 2009-10-08 00:00:00 |

| 7 | Muffy | 1500 | 2009-10-08 00:00:00 |

| 7 | Muffy | 1560 | 2009-11-20 00:00:00 |

| 7 | Muffy | 2060 | 2008-05-20 00:00:00 |

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

# SQL - UNIONS CLAUSE

To use this UNION clause, each SELECT statement must have

* The same number of columns selected
* The same number of column expressions
* The same data type and
* Have them in the same order

## *Syntax*

|  |
| --- |
| SELECT column1 [, column2 ]  FROM table1 [, table2 ]  [WHERE condition]  UNION  SELECT column1 [, column2 ]  FROM table1 [, table2 ]  [WHERE condition] |

## *Example*

Table 1 − CUSTOMERS Table is as follows.

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

| 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 |

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

Table 2 − ORDERS Table is as follows.

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

|OID | DATE | CUSTOMER\_ID | AMOUNT |

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

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

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

join these two tables in our SELECT statement

SQL> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

LEFT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID

UNION

SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

RIGHT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

result

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

| ID | NAME | AMOUNT | DATE |

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

| 1 | Ramesh | NULL | NULL |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

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

## The UNION ALL Clause

The UNION ALL operator is used to combine the results of two SELECT statements including duplicate rows.

The same rules that apply to the UNION clause will apply to the UNION ALL operator.

***Syntax***

|  |
| --- |
| SELECT column1 [, column2 ]  FROM table1 [, table2 ]  [WHERE condition]  UNION ALL  SELECT column1 [, column2 ]  FROM table1 [, table2 ]  [WHERE condition] |

.***Example***

Table 1 − CUSTOMERS Table is as follows.

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

| 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 |

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

Table 2 − ORDERS table is as follows.

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

|OID | DATE | CUSTOMER\_ID | AMOUNT |

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

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

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

join these two tables in our SELECT statement

SQL> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

LEFT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID

UNION ALL

SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

RIGHT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

result

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

| ID | NAME | AMOUNT | DATE |

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

| 1 | Ramesh | NULL | NULL |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

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

There are two other clauses (i.e., operators), which are like the UNION clause.

* SQL [INTERSECT Clause](https://www.tutorialspoint.com/sql/sql-intersect-clause.htm) − This is used to combine two SELECT statements, but returns rows only from the first SELECT statement that are identical to a row in the second SELECT statement.
* SQL [EXCEPT Clause](https://www.tutorialspoint.com/sql/sql-except-clause.htm) − This combines two SELECT statements and returns rows from the first SELECT statement that are not returned by the second SELECT statement.

# SQL - INTERSECT Clause

***Syntax***

|  |
| --- |
| SELECT column1 [, column2 ]  FROM table1 [, table2 ]  [WHERE condition]  INTERSECT  SELECT column1 [, column2 ]  FROM table1 [, table2 ]  [WHERE condition] |

Table 1 − CUSTOMERS Table is as follows

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

| 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 |

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

Table 2 − ORDERS Table is as follows.

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

|OID | DATE | CUSTOMER\_ID | AMOUNT |

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

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

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

join these two tables in our SELECT statement.

SQL> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

LEFT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID

INTERSECT

SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

RIGHT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

result.

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

| ID | NAME | AMOUNT | DATE |

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

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Ramesh | 1560 | 2009-11-20 00:00:00 |

| 4 | kaushik | 2060 | 2008-05-20 00:00:00 |

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

# SQL - EXCEPT Clause

***Syntax***

|  |
| --- |
| SELECT column1 [, column2 ]  FROM table1 [, table2 ]  [WHERE condition]  EXCEPT  SELECT column1 [, column2 ]  FROM table1 [, table2 ]  [WHERE condition] |

***Example***

Table 1 − CUSTOMERS Table is as follows.

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

| 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 |

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

Table 2 − ORDERS table is as follows.

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

|OID | DATE | CUSTOMER\_ID | AMOUNT |

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

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

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

join these two tables in our SELECT statement

SQL> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

LEFT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID

EXCEPT

SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

RIGHT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

result.

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

| ID | NAME | AMOUNT | DATE |

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

| 1 | Ramesh | NULL | NULL |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

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

# SQL - NULL Values

***Syntax***

|  |
| --- |
| 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)); |

**NOT NULL** signifies that column should always accept an explicit value of the given data type. There are two columns where we did not use NOT NULL, which means these columns could be NULL.

A field with a NULL value is the one that has been left blank during the record creation.

***Example***

The NULL value can cause problems when selecting data. However, because when comparing an unknown value to any other value, the result is always unknown and not included in the results. You must use the **IS NULL** or **IS NOT NULL** operators to check for a NULL value.

CUSTOMERS table.

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

| 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 | |

| 7 | Muffy | 24 | Indore | |

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

usage of **IS NOT NULL**operator.

SQL> SELECT ID, NAME, AGE, ADDRESS, SALARY

FROM CUSTOMERS

WHERE SALARY IS NOT NULL;

result

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

| 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 |

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

usage of **IS NULL** operator.

SQL> SELECT ID, NAME, AGE, ADDRESS, SALARY

FROM CUSTOMERS

WHERE SALARY IS NULL;

result −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 6 | Komal | 22 | MP | |

| 7 | Muffy | 24 | Indore | |

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

# SQL - Alias *Syntax*

***Syntax***

|  |
| --- |
| SELECT column1, column2....  FROM table\_name AS alias\_name  WHERE [condition]; |

The basic ***Syntax*** of a **table** alias

|  |
| --- |
| SELECT column\_name AS alias\_name  FROM table\_name  WHERE [condition]; |

The basic ***Syntax*** of a **column** alias

***Example***

Table 1 − CUSTOMERS Table is as follows.

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

| 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 |

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

Table 2 − ORDERS Table is as follows.

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

|OID | DATE | CUSTOMER\_ID | AMOUNT |

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

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

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

usage of a **table alias**.

SQL> SELECT C.ID, C.NAME, C.AGE, O.AMOUNT

FROM CUSTOMERS AS C, ORDERS AS O

WHERE C.ID = O.CUSTOMER\_ID;

result.

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

| ID | NAME | AGE | AMOUNT |

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

| 3 | kaushik | 23 | 3000 |

| 3 | kaushik | 23 | 1500 |

| 2 | Khilan | 25 | 1560 |

| 4 | Chaitali | 25 | 2060 |

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

usage of a **column alias**.

SQL> SELECT ID AS CUSTOMER\_ID, NAME AS CUSTOMER\_NAME

FROM CUSTOMERS

WHERE SALARY IS NOT NULL;

result.

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

| CUSTOMER\_ID | CUSTOMER\_NAME |

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

| 1 | Ramesh |

| 2 | Khilan |

| 3 | kaushik |

| 4 | Chaitali |

| 5 | Hardik |

| 6 | Komal |

| 7 | Muffy |

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

# SQL - Indexes

The following guidelines indicate when the use of an index should be reconsidered.

* Indexes should not be used on small tables.
* Tables that have frequent, large batch updates or insert operations.
* Indexes should not be used on columns that contain a high number of NULL values.
* Columns that are frequently manipulated should not be indexed.

The CREATE INDEX Command

***Syntax***

|  |
| --- |
| CREATE INDEX index\_name ON table\_name; |

Single-Column Indexes

***Syntax***

|  |
| --- |
| CREATE INDEX index\_name ON table\_name (column\_name); |

Unique Indexes

Unique indexes are used not only for performance, but also for data integrity. A unique index does not allow any duplicate values to be inserted into the table.

***Syntax***

|  |
| --- |
| CREATE UNIQUE INDEX index\_name on table\_name (column\_name); |

Composite Indexes

A composite index is an index on two or more columns of a table.

***Syntax***

|  |
| --- |
| CREATE INDEX index\_name on table\_name (column1, column2); |

Implicit Indexes

Implicit indexes are indexes that are automatically created by the database server when an object is created. Indexes are automatically created for primary key constraints and unique constraints.

The DROP INDEX Command

An index can be dropped using SQL **DROP** command. Care should be taken when dropping an index because the performance may either slow down or improve.

***Syntax***

|  |
| --- |
| DROP INDEX index\_name; |

# SQL - ALTER TABLE Command

***Syntax*** for ALTER TABLE command to add a **New Column** in an existing table

|  |
| --- |
| ALTER TABLE table\_name ADD column\_name datatype; |

***Syntax*** for ALTER TABLE command to **DROP COLUMN** in an existing table.

|  |
| --- |
| ALTER TABLE table\_name DROP COLUMN column\_name; |

***Syntax*** for ALTER TABLE command to change the **DATA TYPE** of a column in a table.

|  |
| --- |
| ALTER TABLE table\_name MODIFY COLUMN column\_name datatype; |

***Syntax*** for ALTER TABLE command to add a **NOT NULL** constraint to a column in a table.

|  |
| --- |
| ALTER TABLE table\_name MODIFY column\_name datatype NOT NULL; |

***Syntax*** of ALTER TABLE to **ADD UNIQUE CONSTRAINT** to a table.

|  |
| --- |
| ALTER TABLE table\_name  ADD CONSTRAINT MyUniqueConstraint UNIQUE(column1, column2...); |

***Syntax*** of an ALTER TABLE command to **ADD CHECK CONSTRAINT** to a table.

|  |
| --- |
| ALTER TABLE table\_name  ADD CONSTRAINT MyUniqueConstraint CHECK (CONDITION); |

***Syntax*** of an ALTER TABLE command to **ADD PRIMARY KEY** constraint to a table

|  |
| --- |
| ALTER TABLE table\_name  ADD CONSTRAINT MyPrimaryKey PRIMARY KEY (column1, column2...); |

***Syntax*** of an ALTER TABLE command to **DROP CONSTRAINT** from a table

|  |
| --- |
| ALTER TABLE table\_name  DROP CONSTRAINT MyUniqueConstraint; |

If you're using MySQL

ALTER TABLE table\_name

DROP INDEX MyUniqueConstraint;

***Syntax*** for ALTER TABLE command to **DROP PRIMARY KEY** constraint from a table.

|  |
| --- |
| ALTER TABLE table\_name  DROP CONSTRAINT MyPrimaryKey; |

If you're using MySQL

ALTER TABLE table\_name

DROP PRIMARY KEY;

***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 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

***Example*** to ADD a **New Column** to an existing table −

ALTER TABLE CUSTOMERS ADD SEX char(1);

CUSTOMERS table is changed and following would be output from the SELECT statement.

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

| ID | NAME | AGE | ADDRESS | SALARY | SEX |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 | NULL |

| 2 | Ramesh | 25 | Delhi | 1500.00 | NULL |

| 3 | kaushik | 23 | Kota | 2000.00 | NULL |

| 4 | kaushik | 25 | Mumbai | 6500.00 | NULL |

| 5 | Hardik | 27 | Bhopal | 8500.00 | NULL |

| 6 | Komal | 22 | MP | 4500.00 | NULL |

| 7 | Muffy | 24 | Indore | 10000.00 | NULL |

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

***Example*** to DROP sex column from the existing table.

ALTER TABLE CUSTOMERS DROP SEX;

CUSTOMERS table is changed and following would be the output from the SELECT statement.

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Ramesh | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | kaushik | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

# SQL - TRUNCATE TABLE Command

***Syntax***

The basic ***Syntax*** of a **TRUNCATE TABLE** command is as follows.

|  |
| --- |
| TRUNCATE TABLE table\_name; |

***Example***

Consider a 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 |

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

Following is the ***Example*** of a Truncate command.

|  |
| --- |
| SQL > TRUNCATE TABLE CUSTOMERS; |

Now, the CUSTOMERS table is truncated and the output from SELECT statement will be as shown in the code block below −

|  |
| --- |
| SQL> SELECT \* FROM CUSTOMERS;  Empty set (0.00 sec) |

# SQL - Using Views

Creating Views

Database views are created using the **CREATE VIEW** statement.

To create a view, a user must have the appropriate system privilege according to the specific implementation.

The basic **CREATE VIEW** ***Syntax*** is as follows −

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

You can include multiple tables in your SELECT statement

***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 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

Following is an ***Example*** to create a view from the CUSTOMERS table. This view would be used to have customer name and age from the CUSTOMERS table.

|  |
| --- |
| SQL > CREATE VIEW CUSTOMERS\_VIEW AS  SELECT name, age  FROM CUSTOMERS; |

Now, you can query CUSTOMERS\_VIEW in a similar way as you query an actual table. Following is an ***Example*** for the same.

|  |
| --- |
| SQL > SELECT \* FROM CUSTOMERS\_VIEW; |

This would produce the following result.

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

| name | age |

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

| Ramesh | 32 |

| Khilan | 25 |

| kaushik | 23 |

| Chaitali | 25 |

| Hardik | 27 |

| Komal | 22 |

| Muffy | 24 |

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

The WITH CHECK OPTION

The WITH CHECK OPTION is a CREATE VIEW statement option. The purpose of the WITH CHECK OPTION is to ensure that all UPDATE and INSERTs satisfy the condition(s) in the view definition.

If they do not satisfy the condition(s), the UPDATE or INSERT returns an error.

***Example***:CUSTOMERS\_VIEW with the WITH CHECK OPTION.

|  |
| --- |
| CREATE VIEW CUSTOMERS\_VIEW AS  SELECT name, age  FROM CUSTOMERS  WHERE age IS NOT NULL  WITH CHECK OPTION; |

The WITH CHECK OPTION in this case should deny the entry of any NULL values in the view's AGE column, because the view is defined by data that does not have a NULL value in the AGE column.

Updating a View

A view can be updated under certain conditions which are given below −

* The SELECT clause may not contain the keyword DISTINCT.
* The SELECT clause may not contain summary functions.
* The SELECT clause may not contain set functions.
* The SELECT clause may not contain set operators.
* The SELECT clause may not contain an ORDER BY clause.
* The FROM clause may not contain multiple tables.
* The WHERE clause may not contain subqueries.
* The query may not contain GROUP BY or HAVING.
* Calculated columns may not be updated.
* All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.

***Example*** to update the age of Ramesh.

|  |
| --- |
| SQL > UPDATE CUSTOMERS\_VIEW  SET AGE = 35  WHERE name = 'Ramesh'; |

This would ultimately update the base table CUSTOMERS and the same would reflect in the view itself. Now, try to query the base table and the SELECT statement would produce the following result.

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 35 | 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 |

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

Inserting Rows into a View

Rows of data can be inserted into a view. command also apply to the INSERT command.

Here, we cannot insert rows in the CUSTOMERS\_VIEW because we have not included all the NOT NULL columns in this view, otherwise you can insert rows in a view in a similar way as you insert them in a table.

Deleting Rows into a View

Rows of data can be deleted from a view.

Following is an ***Example*** to delete a record having AGE = 22.

|  |
| --- |
| SQL > DELETE FROM CUSTOMERS\_VIEW  WHERE age = 22; |

This would ultimately delete a row from the base table CUSTOMERS and the same would reflect in the view itself. Now, try to query the base table and the SELECT statement would produce the following result.

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 35 | 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 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

Dropping Views

***Syntax***:To drop the view if it is no longer needed.

|  |
| --- |
| DROP VIEW view\_name; |

***Example***:To drop the CUSTOMERS\_VIEW from the CUSTOMERS table.

|  |
| --- |
| DROP VIEW CUSTOMERS\_VIEW; |

# SQL - Having Clause

The **HAVING Clause** enables you to specify conditions that filter which group results appear in the results.

The WHERE clause places conditions on the selected columns, whereas the HAVING clause places conditions on groups created by the GROUP BY clause.

***Syntax***

The following code block shows the position of the HAVING Clause in a query.

|  |
| --- |
| SELECT  FROM  WHERE  GROUP BY  HAVING  ORDER BY |

***Syntax*** of the SELECT statement including the HAVING clause −

|  |
| --- |
| SELECT column1, column2  FROM table1, table2  WHERE [ conditions ]  GROUP BY column1, column2  HAVING [ conditions ]  ORDER BY column1, column2 |

***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 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

***Example***, which would display a record for a similar age count that would be more than or equal to 2.

SQL > SELECT ID, NAME, AGE, ADDRESS, SALARY

FROM CUSTOMERS

GROUP BY age

HAVING COUNT(age) >= 2;

This would produce the following result −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 2 | Khilan | 25 | Delhi | 1500.00 |

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

# SQL - Transactions

Properties of Transactions

**ACID**.

* **Atomicity** − ensures that all operations within the work unit are completed successfully. Otherwise, the transaction is aborted at the point of failure and all the previous operations are rolled back to their former state.
* **Consistency** − ensures that the database properly changes states upon a successfully committed transaction.
* **Isolation** − enables transactions to operate independently of and transparent to each other.
* **Durability** − ensures that the result or effect of a committed transaction persists in case of a system failure.

Transaction Control

* **COMMIT** − to save the changes.
* **ROLLBACK** − to roll back the changes.
* **SAVEPOINT** − creates points within the groups of transactions in which to ROLLBACK.
* **SET TRANSACTION** − Places a name on a transaction.

Transactional Control Commands

Transactional control commands are only used with the **DML Commands** such as - INSERT, UPDATE and DELETE only.

They cannot be used while creating tables or dropping them because these operations are automatically committed in the database.

The COMMIT Command

The COMMIT command is used to save changes invoked by a transaction to the database.

The COMMIT command saves all the transactions to the database since the last COMMIT or ROLLBACK command.

***Syntax***:for the COMMIT command is as follows.

|  |
| --- |
| COMMIT; |

***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 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

***Example*** to delete those records from the table which have age = 25 and then COMMIT the changes in the database.

SQL> DELETE FROM CUSTOMERS

WHERE AGE = 25;

SQL> COMMIT;

Thus, two rows from the table would be deleted and the SELECT statement would produce the following result.

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

The ROLLBACK Command

The ROLLBACK command is used to undo transactions that have not already been saved to the database.

This command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

***Syntax*** for a ROLLBACK command is as follows −

|  |
| --- |
| ROLLBACK; |

***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 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

***Example***, to delete those records from the table which have the age = 25 and then ROLLBACK the changes in the database.

SQL> DELETE FROM CUSTOMERS

WHERE AGE = 25;

SQL> ROLLBACK;

Thus, the delete operation would not impact the table and the SELECT statement would produce the following result.

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

| 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 SAVEPOINT Command

A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.

***Syntax*** for a SAVEPOINT command is as shown below.

|  |
| --- |
| SAVEPOINT SAVEPOINT\_NAME; |

This command serves only in the creation of a SAVEPOINT among all the transactional statements.

The ROLLBACK command is used to undo a group of transactions.

***Syntax*** for rolling back to a SAVEPOINT is as shown below.

|  |
| --- |
| ROLLBACK TO SAVEPOINT\_NAME; |

***Example*** where you plan to delete the three different records from the CUSTOMERS table. You want to create a SAVEPOINT before each delete, so that you can ROLLBACK to any SAVEPOINT at any time to return the appropriate data to its original state.

***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 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

The following code block contains the series of operations.

SQL> SAVEPOINT SP1;

Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=1;

1 row deleted.

SQL> SAVEPOINT SP2;

Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=2;

1 row deleted.

SQL> SAVEPOINT SP3;

Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=3;

1 row deleted.

Now that the three deletions have taken place, let us assume that you have changed your mind and decided to ROLLBACK to the SAVEPOINT that you identified as SP2. Because SP2 was created after the first deletion, the last two deletions are undone −

SQL> ROLLBACK TO SP2;

Rollback complete.

Notice that only the first deletion took place since you rolled back to SP2.

SQL> SELECT \* FROM CUSTOMERS;

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 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 |

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

6 rows selected.

The RELEASE SAVEPOINT Command

The RELEASE SAVEPOINT command is used to remove a SAVEPOINT that you have created.

The ***Syntax*** for a RELEASE SAVEPOINT command is as follows.

|  |
| --- |
| RELEASE SAVEPOINT SAVEPOINT\_NAME; |

Once a SAVEPOINT has been released, you can no longer use the ROLLBACK command to undo transactions performed since the last SAVEPOINT.

The SET TRANSACTION Command

The SET TRANSACTION command can be used to initiate a database transaction. This command is used to specify characteristics for the transaction that follows. For ***Example***, you can specify a transaction to be read only or read write.

The ***Syntax*** for a SET TRANSACTION command is as follows.

|  |
| --- |
| SET TRANSACTION [ READ WRITE | READ ONLY ]; |

# SQL - Wildcard Operators

|  |  |
| --- | --- |
| **Sr.No.** | **Wildcard & Description** |
| 1 | **The percent sign (%)**  Matches one or more characters.  ***Note*** − MS Access uses the asterisk (\*) wildcard character instead of the percent sign (%) wildcard character. |
| 2 | **The underscore (\_)**  Matches one character.  ***Note*** − MS Access uses a question mark (?) instead of the underscore (\_) to match any one character. |

The percent sign represents zero, one or multiple characters.

The underscore represents a single number or a character.

***Syntax***

The basic ***Syntax*** of a '%' and a '\_' operator is as follows.

|  |
| --- |
| SELECT FROM table\_name  WHERE column LIKE 'XXXX%' (or) |
| SELECT FROM table\_name  WHERE column LIKE '%XXXX%' (or) |
| SELECT FROM table\_name  WHERE column LIKE 'XXXX\_' (or) |
| SELECT FROM table\_name  WHERE column LIKE '\_XXXX' (or) |
| SELECT FROM table\_name  WHERE column LIKE '\_XXXX\_' |

You can combine N number of conditions using the AND or the OR operators. Here, XXXX could be any numeric or string value.

***Example***

the WHERE part having different LIKE clauses with '%' and '\_' operators.

|  |  |
| --- | --- |
| **Sr.No.** | **Statement & Description** |
| 1 | **WHERE SALARY LIKE '200%'**  Finds any values that start with 200. |
| 2 | **WHERE SALARY LIKE '%200%'**  Finds any values that have 200 in any position. |
| 3 | **WHERE SALARY LIKE '\_00%'**  Finds any values that have 00 in the second and third positions. |
| 4 | **WHERE SALARY LIKE '2\_%\_%'**  Finds any values that start with 2 and are at least 3 characters in length. |
| 5 | **WHERE SALARY LIKE '%2'**  Finds any values that end with 2. |
| 6 | **WHERE SALARY LIKE '\_2%3'**  Finds any values that have a 2 in the second position and end with a 3. |
| 7 | **WHERE SALARY LIKE '2\_\_\_3'**  Finds any values in a five-digit number that start with 2 and end with 3. |

***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 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

***Example***,display all the records from the CUSTOMERS table where the SALARY starts with 200.

SQL> SELECT \* FROM CUSTOMERS

WHERE SALARY LIKE '200%';

This would produce the following result.

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

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

# SQL - Date Functions

|  |  |
| --- | --- |
| **Sr.No.** | **Function & Description** |
| 1 | [ADDDATE()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_adddate)  Adds dates |
| 2 | [ADDTIME()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_addtime)  Adds time |
| 3 | [CONVERT\_TZ()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_convert-tz)  Converts from one timezone to another |
| 4 | [CURDATE()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_curdate)  Returns the current date |
| 5 | [CURRENT\_DATE(), CURRENT\_DATE](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_current-date)  Synonyms for CURDATE() |
| 6 | [CURRENT\_TIME(), CURRENT\_TIME](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_current-time)  Synonyms for CURTIME() |
| 7 | [CURRENT\_TIMESTAMP(), CURRENT\_TIMESTAMP](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_current-timestamp)  Synonyms for NOW() |
| 8 | [CURTIME()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_curtime)  Returns the current time |
| 9 | [DATE\_ADD()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_date-add)  Adds two dates |
| 10 | [DATE\_FORMAT()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_date-format)  Formats date as specified |
| 11 | [DATE\_SUB()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_date-sub)  Subtracts two dates |
| 12 | [DATE()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_date)  Extracts the date part of a date or datetime expression |
| 13 | [DATEDIFF()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_datediff)  Subtracts two dates |
| 14 | [DAY()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_day)  Synonym for DAYOFMONTH() |
| 15 | [DAYNAME()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_dayname)  Returns the name of the weekday |
| 16 | [DAYOFMONTH()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_dayofmonth)  Returns the day of the month (1-31) |
| 17 | [DAYOFWEEK()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_dayofweek)  Returns the weekday index of the argument |
| 18 | [DAYOFYEAR()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_dayofyear)  Returns the day of the year (1-366) |
| 19 | [EXTRACT](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_extract)  Extracts part of a date |
| 20 | [FROM\_DAYS()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_from-days)  Converts a day number to a date |
| 21 | [FROM\_UNIXTIME()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_from-unixtime)  Formats date as a UNIX timestamp |
| 22 | [HOUR()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_hour)  Extracts the hour |
| 23 | [LAST\_DAY](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_last-day)  Returns the last day of the month for the argument |
| 24 | [LOCALTIME(), LOCALTIME](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_localtime)  Synonym for NOW() |
| 25 | [LOCALTIMESTAMP, LOCALTIMESTAMP()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_localtimestamp)  Synonym for NOW() |
| 26 | [MAKEDATE()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_makedate)  Creates a date from the year and day of year |
| 27 | [MAKETIME](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_maketime)  MAKETIME() |
| 28 | [MICROSECOND()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_microsecond)  Returns the microseconds from argument |
| 29 | [MINUTE()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_minute)  Returns the minute from the argument |
| 30 | [MONTH()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_month)  Return the month from the date passed |
| 31 | [MONTHNAME()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_monthname)  Returns the name of the month |
| 32 | [NOW()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_now)  Returns the current date and time |
| 33 | [PERIOD\_ADD()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_period-add)  Adds a period to a year-month |
| 34 | [PERIOD\_DIFF()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_period-diff)  Returns the number of months between periods |
| 35 | [QUARTER()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_quarter)  Returns the quarter from a date argument |
| 36 | [SEC\_TO\_TIME()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_sec-to-time)  Converts seconds to 'HH:MM:SS' format |
| 37 | [SECOND()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_second)  Returns the second (0-59) |
| 38 | [STR\_TO\_DATE()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_str-to-date)  Converts a string to a date |
| 39 | [SUBDATE()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_subdate)  When invoked with three arguments a synonym for DATE\_SUB() |
| 40 | [SUBTIME()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_subtime)  Subtracts times |
| 41 | [SYSDATE()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_sysdate)  Returns the time at which the function executes |
| 42 | [TIME\_FORMAT()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_time-format)  Formats as time |
| 43 | [TIME\_TO\_SEC()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_time-to-sec)  Returns the argument converted to seconds |
| 44 | [TIME()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_time)  Extracts the time portion of the expression passed |
| 45 | [TIMEDIFF()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_timediff)  Subtracts time |
| 46 | [TIMESTAMP()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_timestamp)  With a single argument this function returns the date or datetime expression. With two arguments, the sum of the arguments |
| 47 | [TIMESTAMPADD()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_timestampadd)  Adds an interval to a datetime expression |
| 48 | [TIMESTAMPDIFF()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_timestampdiff)  Subtracts an interval from a datetime expression |
| 49 | [TO\_DAYS()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_to-days)  Returns the date argument converted to days |
| 50 | [UNIX\_TIMESTAMP()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_unix-timestamp)  Returns a UNIX timestamp |
| 51 | [UTC\_DATE()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_utc-date)  Returns the current UTC date |
| 52 | [UTC\_TIME()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_utc-time)  Returns the current UTC time |
| 53 | [UTC\_TIMESTAMP()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_utc-timestamp)  Returns the current UTC date and time |
| 54 | [WEEK()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_week)  Returns the week number |
| 55 | [WEEKDAY()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_weekday)  Returns the weekday index |
| 56 | [WEEKOFYEAR()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_weekofyear)  Returns the calendar week of the date (1-53) |
| 57 | [YEAR()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_year)  Returns the year |
| 58 | [YEARWEEK()](https://www.tutorialspoint.com/sql/sql-date-functions.htm#function_yearweek)  Returns the year and week |

## 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)

## CONVERT\_TZ(dt,from\_tz,to\_tz)

This converts a datetime value dt from the time zone given by from\_tz to the time zone given by to\_tz and returns the resulting value. This function returns NULL if the arguments are invalid.

mysql> SELECT CONVERT\_TZ('2004-01-01 12:00:00','GMT','MET');

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

| CONVERT\_TZ('2004-01-01 12:00:00','GMT','MET') |

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

| 2004-01-01 13:00:00 |

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

1 row in set (0.00 sec)

mysql> SELECT CONVERT\_TZ('2004-01-01 12:00:00','+00:00','+10:00');

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

| CONVERT\_TZ('2004-01-01 12:00:00','+00:00','+10:00') |

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

| 2004-01-01 22:00:00 |

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

1 row in set (0.00 sec)

## CURDATE()

Returns the current date as a value in 'YYYY-MM-DD' or YYYYMMDD format, depending on whether the function is used in a string or in a numeric context.

mysql> SELECT CURDATE();

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

| CURDATE() |

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

| 1997-12-15 |

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

1 row in set (0.00 sec)

mysql> SELECT CURDATE() + 0;

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

| CURDATE() + 0 |

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

| 19971215 |

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

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)

## DATEDIFF(expr1,expr2)

DATEDIFF() returns expr1 . expr2 expressed as a value in days from one date to the other. Both expr1 and expr2 are date or date-and-time expressions. Only the date parts of the values are used in the calculation.

mysql> SELECT DATEDIFF('1997-12-31 23:59:59','1997-12-30');

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

| DATEDIFF('1997-12-31 23:59:59','1997-12-30') |

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

| 1 |

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

1 row in set (0.00 sec)

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

These functions perform date arithmetic.

The **date** is a DATETIME or DATE value specifying the starting date.

The **expr** is an expression specifying the interval value to be added or subtracted from the starting date.

The expr is a string; it may start with a '-' for negative intervals.

A **unit** is a keyword indicating the units in which the expression should be interpreted.

The **INTERVAL** keyword and the unit specifier are not case sensitive.

The following table shows the expected form of the expr argument for each unit value.

|  |  |
| --- | --- |
| **unit Value** | **Expected exprFormat** |
| MICROSECOND | MICROSECONDS |
| SECOND | SECONDS |
| MINUTE | MINUTES |
| HOUR | HOURS |
| DAY | DAYS |
| WEEK | WEEKS |
| MONTH | MONTHS |
| QUARTER | QUARTERS |
| YEAR | YEARS |
| SECOND\_MICROSECOND | 'SECONDS.MICROSECONDS' |
| MINUTE\_MICROSECOND | 'MINUTES.MICROSECONDS' |
| MINUTE\_SECOND | 'MINUTES:SECONDS' |
| HOUR\_MICROSECOND | 'HOURS.MICROSECONDS' |
| HOUR\_SECOND | 'HOURS:MINUTES:SECONDS' |
| HOUR\_MINUTE | 'HOURS:MINUTES' |
| DAY\_MICROSECOND | 'DAYS.MICROSECONDS' |
| DAY\_SECOND | 'DAYS HOURS:MINUTES:SECONDS' |
| DAY\_MINUTE | 'DAYS HOURS:MINUTES' |
| DAY\_HOUR | 'DAYS HOURS' |
| YEAR\_MONTH | 'YEARS-MONTHS' |

The values **QUARTER** and **WEEK** are available from the MySQL 5.0.0. version.

mysql> SELECT DATE\_ADD('1997-12-31 23:59:59',

-> INTERVAL '1:1' MINUTE\_SECOND);

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

| DATE\_ADD('1997-12-31 23:59:59', INTERVAL... |

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

| 1998-01-01 00:01:00 |

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

1 row in set (0.00 sec)

mysql> SELECT DATE\_ADD('1999-01-01', INTERVAL 1 HOUR);

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

| DATE\_ADD('1999-01-01', INTERVAL 1 HOUR) |

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

| 1999-01-01 01:00:00 |

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

1 row in set (0.00 sec)

## DATE\_FORMAT(date,format)

This command formats the date value as per 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().

## MAKEDATE(year,dayofyear)

Returns a date, given year and day-of-year values. The dayofyear value must be greater than 0 or the result will be NULL.

mysql> SELECT MAKEDATE(2001,31), MAKEDATE(2001,32);

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

| MAKEDATE(2001,31), MAKEDATE(2001,32) |

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

| '2001-01-31', '2001-02-01' |

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

1 row in set (0.00 sec)

## MAKETIME(hour,minute,second)

Returns a time value calculated from the hour, minute and second arguments.

mysql> SELECT MAKETIME(12,15,30);

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

| MAKETIME(12,15,30) |

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

| '12:15:30' |

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

1 row in set (0.00 sec)

## MICROSECOND(expr)

Returns the microseconds from the time or datetime expression (expr) as a number in the range from 0 to 999999.

mysql> SELECT MICROSECOND('12:00:00.123456');

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

| MICROSECOND('12:00:00.123456') |

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

| 123456 |

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

1 row in set (0.00 sec)

## MINUTE(time)

Returns the minute for time, in the range 0 to 59.

mysql> SELECT MINUTE('98-02-03 10:05:03');

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

| MINUTE('98-02-03 10:05:03') |

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

| 5 |

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

1 row in set (0.00 sec)

## MONTH(date)

Returns the month for date, in the range 0 to 12.

mysql> SELECT MONTH('1998-02-03')

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

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

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

| 2 |

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

1 row in set (0.00 sec)

## MONTHNAME(date)

Returns the full name of the month for a date.

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

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

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

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

| February |

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

1 row in set (0.00 sec)

## NOW()

Returns the current date and time as a value in 'YYYY-MM-DD HH:MM:SS' or YYYYMMDDHHMMSS format, depending on whether the function is used in a string or numeric context. This value is expressed in the current time zone.

mysql> SELECT NOW();

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

| NOW() |

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

| 1997-12-15 23:50:26 |

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

1 row in set (0.00 sec)

## PERIOD\_ADD(P,N)

Adds N months to a period P (in the format YYMM or YYYYMM). Returns a value in the format YYYYMM. ***Note*** that the period argument P is not a date value.

mysql> SELECT PERIOD\_ADD(9801,2);

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

| PERIOD\_ADD(9801,2) |

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

| 199803 |

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

1 row in set (0.00 sec)

## PERIOD\_DIFF(P1,P2)

Returns the number of months between periods P1 and P2. These periods P1 and P2 should be in the format YYMM or YYYYMM. ***Note*** that the period arguments P1 and P2 are not date values.

mysql> SELECT PERIOD\_DIFF(9802,199703);

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

| PERIOD\_DIFF(9802,199703) |

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

| 11 |

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

1 row in set (0.00 sec)

## QUARTER(date)

Returns the quarter of the year for date, in the range 1 to 4.

mysql> SELECT QUARTER('98-04-01');

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

| QUARTER('98-04-01') |

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

| 2 |

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

1 row in set (0.00 sec)

## SECOND(time)

Returns the second for time, in the range 0 to 59.

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

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

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

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

| 3 |

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

1 row in set (0.00 sec)

## SEC\_TO\_TIME(seconds)

Returns the seconds argument, converted to hours, minutes and seconds, as a value in 'HH:MM:SS' or HHMMSS format, depending on whether the function is used in a string or numeric context.

mysql> SELECT SEC\_TO\_TIME(2378);

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

| SEC\_TO\_TIME(2378) |

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

| 00:39:38 |

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

1 row in set (0.00 sec)

## STR\_TO\_DATE(str,format)

This is the inverse of the DATE\_FORMAT() function. It takes a string str and a format string format. The STR\_TO\_DATE() function returns a DATETIME value if the format string contains both date and time parts. Else, it returns a DATE or TIME value if the string contains only date or time parts.

mysql> SELECT STR\_TO\_DATE('04/31/2004', '%m/%d/%Y');

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

| STR\_TO\_DATE('04/31/2004', '%m/%d/%Y') |

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

| 2004-04-31 |

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

1 row in set (0.00 sec)

## SUBDATE(date,INTERVAL expr unit) and SUBDATE(expr,days)

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

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

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

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

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

| 1997-12-02 |

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

1 row in set (0.00 sec)

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

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

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

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

| 1997-12-02 |

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

1 row in set (0.00 sec)

## SUBTIME(expr1,expr2)

The SUBTIME() function returns expr1 . expr2 expressed as a value in the same format as expr1. The expr1 value is a time or a datetime expression, while the expr2 value is a time expression.

mysql> SELECT SUBTIME('1997-12-31 23:59:59.999999',

-> '1 1:1:1.000002');

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

| SUBTIME('1997-12-31 23:59:59.999999'... |

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

| 1997-12-30 22:58:58.999997 |

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

1 row in set (0.00 sec)

## SYSDATE()

Returns the current date and time 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.

mysql> SELECT SYSDATE();

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

| SYSDATE() |

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

| 2006-04-12 13:47:44 |

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

1 row in set (0.00 sec)

## TIME(expr)

Extracts the time part of the time or datetime expression **expr** and returns it as a string.

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

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

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

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

| 01:02:03 |

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

1 row in set (0.00 sec)

## TIMEDIFF(expr1,expr2)

The TIMEDIFF() function returns expr1 . expr2 expressed as a time value. These expr1 and expr2 values are time or date-and-time expressions, but both must be of the same type.

mysql> SELECT TIMEDIFF('1997-12-31 23:59:59.000001',

-> '1997-12-30 01:01:01.000002');

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

| TIMEDIFF('1997-12-31 23:59:59.000001'..... |

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

| 46:58:57.999999 |

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

1 row in set (0.00 sec)

## TIMESTAMP(expr), TIMESTAMP(expr1,expr2)

With a single argument, this function returns the date or datetime expression expr as a datetime value. With two arguments, it adds the time expression expr2 to the date or datetime expression **expr1** and returns the result as a datetime value.

mysql> SELECT TIMESTAMP('2003-12-31');

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

| TIMESTAMP('2003-12-31') |

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

| 2003-12-31 00:00:00 |

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

1 row in set (0.00 sec)

## TIMESTAMPADD(unit,interval,datetime\_expr)

This function adds the integer expression interval to the date or datetime expression **datetime\_expr**. The unit for interval is given by the unit argument, which should be one of the following values −

* FRAC\_SECOND
* SECOND, MINUTE
* HOUR, DAY
* WEEK
* MONTH
* QUARTER or
* YEAR

The unit value may be specified using one of the keywords as shown or with a prefix of SQL\_TSI\_.

For ***Example***, DAY and SQL\_TSI\_DAY both are legal.

mysql> SELECT TIMESTAMPADD(MINUTE,1,'2003-01-02');

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

| TIMESTAMPADD(MINUTE,1,'2003-01-02') |

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

| 2003-01-02 00:01:00 |

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

1 row in set (0.00 sec)

## TIMESTAMPDIFF(unit,datetime\_expr1,datetime\_expr2)

Returns the integer difference between the date or datetime expressions datetime\_expr1 and datetime\_expr2. The unit for the result is given by the unit argument. The legal values for the unit are the same as those listed in the description of the TIMESTAMPADD() function.

mysql> SELECT TIMESTAMPDIFF(MONTH,'2003-02-01','2003-05-01');

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

| TIMESTAMPDIFF(MONTH,'2003-02-01','2003-05-01') |

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

| 3 |

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

1 row in set (0.00 sec)

## TIME\_FORMAT(time,format)

This function is used like the DATE\_FORMAT() function, but the format string may contain format specifiers only for hours, minutes and seconds.

If the time value contains an hour part that is greater than 23, the %**H** and %**k** hour format specifiers produce a value larger than the usual range of 0 to 23. The other hour format specifiers produce the hour value modulo 12.

mysql> SELECT TIME\_FORMAT('100:00:00', '%H %k %h %I %l');

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

| TIME\_FORMAT('100:00:00', '%H %k %h %I %l') |

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

| 100 100 04 04 4 |

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

1 row in set (0.00 sec)

## TIME\_TO\_SEC(time)

Returns the time argument converted to seconds.

mysql> SELECT TIME\_TO\_SEC('22:23:00');

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

| TIME\_TO\_SEC('22:23:00') |

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

| 80580 |

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

1 row in set (0.00 sec)

## TO\_DAYS(date)

Given a date, returns a day number (the number of days since year 0).

mysql> SELECT TO\_DAYS(950501);

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

| TO\_DAYS(950501) |

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

| 728779 |

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

1 row in set (0.00 sec)

## UNIX\_TIMESTAMP(), UNIX\_TIMESTAMP(date)

If called with no argument, this function returns a Unix timestamp (seconds since '1970-01-01 00:00:00' UTC) as an unsigned integer. If UNIX\_TIMESTAMP() is called with a date argument, it returns the value of the argument as seconds since '1970-01-01 00:00:00' UTC. date may be a DATE string, a DATETIME string, a TIMESTAMP, or a number in the format YYMMDD or YYYYMMDD.

mysql> SELECT UNIX\_TIMESTAMP();

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

| UNIX\_TIMESTAMP() |

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

| 882226357 |

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

1 row in set (0.00 sec)

mysql> SELECT UNIX\_TIMESTAMP('1997-10-04 22:23:00');

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

| UNIX\_TIMESTAMP('1997-10-04 22:23:00') |

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

| 875996580 |

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

1 row in set (0.00 sec)

## UTC\_DATE, UTC\_DATE()

Returns the current UTC date as a value in 'YYYY-MM-DD' or YYYYMMDD format, depending on whether the function is used in a string or numeric context.

mysql> SELECT UTC\_DATE(), UTC\_DATE() + 0;

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

| UTC\_DATE(), UTC\_DATE() + 0 |

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

| 2003-08-14, 20030814 |

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

1 row in set (0.00 sec)

## UTC\_TIME, UTC\_TIME()

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

mysql> SELECT UTC\_TIME(), UTC\_TIME() + 0;

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

| UTC\_TIME(), UTC\_TIME() + 0 |

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

| 18:07:53, 180753 |

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

1 row in set (0.00 sec)

## UTC\_TIMESTAMP, UTC\_TIMESTAMP()

Returns the current UTC date and time as a value in 'YYYY-MM-DD HH:MM:SS' or in a YYYYMMDDHHMMSS format, depending on whether the function is used in a string or in a numeric context.

mysql> SELECT UTC\_TIMESTAMP(), UTC\_TIMESTAMP() + 0;

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

| UTC\_TIMESTAMP(), UTC\_TIMESTAMP() + 0 |

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

| 2003-08-14 18:08:04, 20030814180804 |

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

1 row in set (0.00 sec)

## WEEK(date[,mode])

This function returns the week number for date. The two-argument form of WEEK() allows you to specify whether the week starts on a Sunday or a Monday and whether the return value should be in the range from 0 to 53 or from 1 to 53. If the mode argument is omitted, the value of the default\_week\_format system variable is used

|  |  |  |  |
| --- | --- | --- | --- |
| **Mode** | **First Day of week** | **Range** | **Week 1 is the first week.** |
| 0 | Sunday | 0-53 | with a Sunday in this year |
| 1 | Monday | 0-53 | with more than 3 days this year |
| 2 | Sunday | 1-53 | with a Sunday in this year |
| 3 | Monday | 1-53 | with more than 3 days this year |
| 4 | Sunday | 0-53 | with more than 3 days this year |
| 5 | Monday | 0-53 | with a Monday in this year |
| 6 | Sunday | 1-53 | with more than 3 days this year |
| 7 | Monday | 1-53 | with a Monday in this year |

mysql> SELECT WEEK('1998-02-20');

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

| WEEK('1998-02-20') |

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

| 7 |

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

1 row in set (0.00 sec)

## WEEKDAY(date)

Returns the weekday index for date (0 = Monday, 1 = Tuesday, . 6 = Sunday).

mysql> SELECT WEEKDAY('1998-02-03 22:23:00');

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

| WEEKDAY('1998-02-03 22:23:00') |

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

| 1 |

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

1 row in set (0.00 sec)

## WEEKOFYEAR(date)

Returns the calendar week of the date as a number in the range from 1 to 53. WEEKOFYEAR() is a compatibility function that is equivalent to WEEK(date,3).

mysql> SELECT WEEKOFYEAR('1998-02-20');

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

| WEEKOFYEAR('1998-02-20') |

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

| 8 |

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

1 row in set (0.00 sec)

## YEAR(date)

Returns the year for date, in the range 1000 to 9999, or 0 for the .zero. date.

mysql> SELECT YEAR('98-02-03');

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

| YEAR('98-02-03') |

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

| 1998 |

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

1 row in set (0.00 sec)

## YEARWEEK(date), YEARWEEK(date,mode)

Returns the year and the week for a date. The mode argument works exactly like the mode argument to the WEEK() function. The year in the result may be different from the year in the date argument for the first and the last week of the year.

mysql> SELECT YEARWEEK('1987-01-01');

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

| YEAR('98-02-03')YEARWEEK('1987-01-01') |

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

| 198653 |

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

1 row in set (0.00 sec)

***Note*** − The week number is different from what the WEEK() function would return (0) for optional arguments 0 or 1, as WEEK() then returns the week in the context of the given year.

# SQL - Temporary Tables

What are Temporary Tables?

There are RDBMS, which support temporary tables. Temporary Tables are a great feature that lets you **store and process intermediate results** by using the same selection, update, and join capabilities that you can use with typical SQL Server tables.

The temporary tables could be very useful in some cases to keep temporary data. The most important thing that should be known for temporary tables is that they will be deleted when the current client session terminates.

Temporary tables are available in MySQL version 3.23 onwards. If you use an older version of MySQL than 3.23, you can't use temporary tables, but you can use **heap tables**.

As stated earlier, temporary tables will only last as long as the session is alive. If you run the code in a PHP script, the temporary table will be destroyed automatically when the script finishes executing. If you are connected to the MySQL database server through the MySQL client program, then the temporary table will exist until you close the client or manually destroy the table.

***Example***

Here is an ***Example*** showing you the usage of a temporary table.

mysql> CREATE TEMPORARY TABLE SALESSUMMARY (

-> product\_name VARCHAR(50) NOT NULL

-> , total\_sales DECIMAL(12,2) NOT NULL DEFAULT 0.00

-> , avg\_unit\_price DECIMAL(7,2) NOT NULL DEFAULT 0.00

-> , total\_units\_sold INT UNSIGNED NOT NULL DEFAULT 0

);

Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO SALESSUMMARY

-> (product\_name, total\_sales, avg\_unit\_price, total\_units\_sold)

-> VALUES

-> ('cucumber', 100.25, 90, 2);

mysql> SELECT \* FROM SALESSUMMARY;

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

| product\_name | total\_sales | avg\_unit\_price | total\_units\_sold |

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

| cucumber | 100.25 | 90.00 | 2 |

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

1 row in set (0.00 sec)

When you issue a SHOW TABLES command, then your temporary table will not be listed out in the list. Now, if you log out of the MySQL session and then issue a SELECT command, you will find no data available in the database. Even your temporary table will not be existing.

Dropping Temporary Tables

By default, all the temporary tables are deleted by MySQL when your database connection gets terminated. Still if you want to delete them in between, then you can do so by issuing a **DROP TABLE** command.

Following is an ***Example*** on dropping a temporary table.

mysql> CREATE TEMPORARY TABLE SALESSUMMARY (

-> product\_name VARCHAR(50) NOT NULL

-> , total\_sales DECIMAL(12,2) NOT NULL DEFAULT 0.00

-> , avg\_unit\_price DECIMAL(7,2) NOT NULL DEFAULT 0.00

-> , total\_units\_sold INT UNSIGNED NOT NULL DEFAULT 0

);

Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO SALESSUMMARY

-> (product\_name, total\_sales, avg\_unit\_price, total\_units\_sold)

-> VALUES

-> ('cucumber', 100.25, 90, 2);

mysql> SELECT \* FROM SALESSUMMARY;

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

| product\_name | total\_sales | avg\_unit\_price | total\_units\_sold |

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

| cucumber | 100.25 | 90.00 | 2 |

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

1 row in set (0.00 sec)

mysql> DROP TABLE SALESSUMMARY;

mysql> SELECT \* FROM SALESSUMMARY;

ERROR 1146: Table 'TUTORIALS.SALESSUMMARY' doesn't exist

# SQL - Clone Tables

* Use SHOW CREATE TABLE command to get a CREATE TABLE statement that specifies the source table's structure, indexes and all.
* Modify the statement to change the table name to that of the clone table and execute the statement. This way you will have an exact clone table.
* Optionally, if you need the table contents copied as well, issue an INSERT INTO or a SELECT statement too.

***Example***

Try out the following ***Example*** to create a clone table for **TUTORIALS\_TBL** whose structure is as follows −

**Step 1** − Get the complete structure about the table.

SQL> SHOW CREATE TABLE TUTORIALS\_TBL \G;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Table: TUTORIALS\_TBL

Create Table: CREATE TABLE 'TUTORIALS\_TBL' (

'tutorial\_id' int(11) NOT NULL auto\_increment,

'tutorial\_title' varchar(100) NOT NULL default '',

'tutorial\_author' varchar(40) NOT NULL default '',

'submission\_date' date default NULL,

PRIMARY KEY ('tutorial\_id'),

UNIQUE KEY 'AUTHOR\_INDEX' ('tutorial\_author')

) TYPE = MyISAM

1 row in set (0.00 sec)

**Step 2** − Rename this table and create another table.

SQL> CREATE TABLE `CLONE\_TBL` (

-> 'tutorial\_id' int(11) NOT NULL auto\_increment,

-> 'tutorial\_title' varchar(100) NOT NULL default '',

-> 'tutorial\_author' varchar(40) NOT NULL default '',

-> 'submission\_date' date default NULL,

-> PRIMARY KEY (`tutorial\_id'),

-> UNIQUE KEY 'AUTHOR\_INDEX' ('tutorial\_author')

-> ) TYPE = MyISAM;

Query OK, 0 rows affected (1.80 sec)

**Step 3** − After executing step 2, you will clone a table in your database. If you want to copy data from an old table, then you can do it by using the INSERT INTO... SELECT statement.

SQL> INSERT INTO CLONE\_TBL (tutorial\_id,

-> tutorial\_title,

-> tutorial\_author,

-> submission\_date)

-> SELECT tutorial\_id,tutorial\_title,

-> tutorial\_author,submission\_date,

-> FROM TUTORIALS\_TBL;

Query OK, 3 rows affected (0.07 sec)

Records: 3 Duplicates: 0 Warnings: 0

Finally, you will have an exact clone table as you wanted to have.

# SQL - Sub Queries

There are a few rules that subqueries must follow −

* Subqueries must be enclosed within parentheses.
* A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
* An ORDER BY command cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY command can be used to perform the same function as the ORDER BY in a subquery.
* Subqueries that return more than one row can only be used with multiple value operators such as the IN operator.
* The SELECT list cannot include any references to values that evaluate to a BLOB, ARRAY, CLOB, or NCLOB.
* A subquery cannot be immediately enclosed in a set function.
* The BETWEEN operator cannot be used with a subquery. However, the BETWEEN operator can be used within the subquery.

Subqueries with the SELECT Statement

Subqueries are most frequently used with the SELECT statement.

***Syntax***

|  |
| --- |
| SELECT column\_name [, column\_name ]  FROM table1 [, table2 ]  WHERE column\_name OPERATOR  (SELECT column\_name [, column\_name ]  FROM table1 [, table2 ]  [WHERE]) |

***Example***

Consider the CUSTOMERS table having the following records −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 35 | 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 |

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

Now, let us check the following subquery with a SELECT statement.

SQL> SELECT \*

FROM CUSTOMERS

WHERE ID IN (SELECT ID

FROM CUSTOMERS

WHERE SALARY > 4500) ;

This would produce the following result.

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

Subqueries with the INSERT Statement

Subqueries also can be used with INSERT statements. The INSERT statement uses the data returned from the subquery to insert into another table. The selected data in the subquery can be modified with any of the character, date or number functions.

***Syntax***

|  |
| --- |
| INSERT INTO table\_name [ (column1 [, column2 ]) ]  SELECT [ \*|column1 [, column2 ]  FROM table1 [, table2 ]  [ WHERE VALUE OPERATOR ] |

***Example***

Consider a table CUSTOMERS\_BKP with similar structure as CUSTOMERS table. Now to copy the complete CUSTOMERS table into the CUSTOMERS\_BKP table, you can use the following ***Syntax***.

|  |
| --- |
| SQL> INSERT INTO CUSTOMERS\_BKP  SELECT \* FROM CUSTOMERS  WHERE ID IN (SELECT ID  FROM CUSTOMERS) ; |

Subqueries with the UPDATE Statement

The subquery can be used in conjunction with the UPDATE statement. Either single or multiple columns in a table can be updated when using a subquery with the UPDATE statement.

***Syntax***.

|  |
| --- |
| UPDATE table  SET column\_name = new\_value  [ WHERE OPERATOR [ VALUE ]  (SELECT COLUMN\_NAME  FROM TABLE\_NAME)  [ WHERE) ] |

***Example***

Assuming, we have CUSTOMERS\_BKP table available which is backup of CUSTOMERS table. The following ***Example*** updates SALARY by 0.25 times in the CUSTOMERS table for all the customers whose AGE is greater than or equal to 27.

SQL> UPDATE CUSTOMERS

SET SALARY = SALARY \* 0.25

WHERE AGE IN (SELECT AGE FROM CUSTOMERS\_BKP

WHERE AGE >= 27 );

This would impact two rows and finally CUSTOMERS table would have the following records.

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 35 | Ahmedabad | 125.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 2125.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

Subqueries with the DELETE Statement

The subquery can be used in conjunction with the DELETE statement like with any other statements mentioned above.

The basic ***Syntax*** is as follows.

|  |
| --- |
| DELETE FROM TABLE\_NAME  [ WHERE OPERATOR [ VALUE ]  (SELECT COLUMN\_NAME  FROM TABLE\_NAME)  [ WHERE) ] |

***Example***

Assuming, we have a CUSTOMERS\_BKP table available which is a backup of the CUSTOMERS table. The following ***Example*** deletes the records from the CUSTOMERS table for all the customers whose AGE is greater than or equal to 27.

SQL> DELETE FROM CUSTOMERS

WHERE AGE IN (SELECT AGE FROM CUSTOMERS\_BKP

WHERE AGE >= 27 );

This would impact two rows and finally the CUSTOMERS table would have the following records.

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

# SQL - Using Sequences

Using AUTO\_INCREMENT column

The simplest way in MySQL to use sequences is to define a column as AUTO\_INCREMENT and leave the rest to MySQL to take care.

***Example***

Try out the following ***Example***. This will create a table and after that it will insert a few rows in this table where it is not required to give a record ID because its auto-incremented by MySQL.

mysql> CREATE TABLE INSECT

-> (

-> id INT UNSIGNED NOT NULL AUTO\_INCREMENT,

-> PRIMARY KEY (id),

-> name VARCHAR(30) NOT NULL, # type of insect

-> date DATE NOT NULL, # date collected

-> origin VARCHAR(30) NOT NULL # where collected

);

Query OK, 0 rows affected (0.02 sec)

mysql> INSERT INTO INSECT (id,name,date,origin) VALUES

-> (NULL,'housefly','2001-09-10','kitchen'),

-> (NULL,'millipede','2001-09-10','driveway'),

-> (NULL,'grasshopper','2001-09-10','front yard');

Query OK, 3 rows affected (0.02 sec)

Records: 3 Duplicates: 0 Warnings: 0

mysql> SELECT \* FROM INSECT ORDER BY id;

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

| id | name | date | origin |

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

| 1 | housefly | 2001-09-10 | kitchen |

| 2 | millipede | 2001-09-10 | driveway |

| 3 | grasshopper | 2001-09-10 | front yard |

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

3 rows in set (0.00 sec)

Obtain AUTO\_INCREMENT Values

The LAST\_INSERT\_ID( ) is an SQL function, so you can use it from within any client that understands how to issue SQL statements. Otherwise PERL and PHP scripts provide exclusive functions to retrieve auto-incremented value of last record.

PERL ***Example***

Use the **mysql\_insertid** attribute to obtain the AUTO\_INCREMENT value generated by a query. This attribute is accessed through either a database handle or a statement handle, depending on how you issue the query. The following ***Example*** references it through the database handle.

$dbh->do ("INSERT INTO INSECT (name,date,origin)

VALUES('moth','2001-09-14','windowsill')");

my $seq = $dbh->{mysql\_insertid};

PHP ***Example***

After issuing a query that generates an AUTO\_INCREMENT value, retrieve the value by calling the **mysql\_insert\_id( )** function.

mysql\_query ("INSERT INTO INSECT (name,date,origin)

VALUES('moth','2001-09-14','windowsill')", $conn\_id);

$seq = mysql\_insert\_id ($conn\_id);

Renumbering an Existing Sequence

There may be a case when you have deleted many records from a table and you want to re-sequence all the records. This can be done by using a simple trick, but you should be very careful to do this and check if your table is having a join with another table or not.

If you determine that resequencing an AUTO\_INCREMENT column is unavoidable, the way to do it is to drop the column from the table, then add it again.

The following ***Example*** shows how to renumber the id values in the insect table using this technique.

mysql> ALTER TABLE INSECT DROP id;

mysql> ALTER TABLE insect

-> ADD id INT UNSIGNED NOT NULL AUTO\_INCREMENT FIRST,

-> ADD PRIMARY KEY (id);

Starting a Sequence at a Particular Value

By default, MySQL will start the sequence from 1, but you can specify any other number as well at the time of table creation.

The following code block has an ***Example*** where MySQL will start sequence from 100.

mysql> CREATE TABLE INSECT

-> (

-> id INT UNSIGNED NOT NULL AUTO\_INCREMENT = 100,

-> PRIMARY KEY (id),

-> name VARCHAR(30) NOT NULL, # type of insect

-> date DATE NOT NULL, # date collected

-> origin VARCHAR(30) NOT NULL # where collected

);

Alternatively, you can create the table and then set the initial sequence value with ALTER TABLE.

mysql> ALTER TABLE t AUTO\_INCREMENT = 100;

# SQL - Handling Duplicates

The SQL **DISTINCT** keyword, which we have already discussed is used in conjunction with the SELECT statement to eliminate all the duplicate records and by fetching only the unique records.

***Syntax***

The basic ***Syntax*** of a DISTINCT keyword to eliminate duplicate records is as follows.

|  |
| --- |
| SELECT DISTINCT column1, column2,.....columnN  FROM table\_name  WHERE [condition] |

***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 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

First, let us see how the following SELECT query returns duplicate salary records.

SQL> SELECT SALARY FROM CUSTOMERS

ORDER BY SALARY;

This would produce the following result where the salary of 2000 is coming twice which is a duplicate record from the original table.

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

| SALARY |

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

| 1500.00 |

| 2000.00 |

| 2000.00 |

| 4500.00 |

| 6500.00 |

| 8500.00 |

| 10000.00 |

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

Now, let us use the DISTINCT keyword with the above SELECT query and see the result.

SQL> SELECT DISTINCT SALARY FROM CUSTOMERS

ORDER BY SALARY;

This would produce the following result where we do not have any duplicate entry.

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

| SALARY |

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

| 1500.00 |

| 2000.00 |

| 4500.00 |

| 6500.00 |

| 8500.00 |

| 10000.00 |

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

# SQL - Injection

If you take a user input through a webpage and insert it into a SQL database, there is a chance that you have left yourself wide open for a security issue known as the **SQL Injection**.

Injection usually occurs when you ask a user for input, like their name and instead of a name they give you a SQL statement that you will unknowingly run on your database. Never trust user provided data, process this data only after validation; as a rule, this is done by **Pattern Matching**.

***Example***,the **name** is restricted to the alphanumerical characters plus underscore and to a length between 8 and 20 characters (modify these rules as needed).

if (preg\_match("/^\w{8,20}$/", $\_GET['username'], $matches)) {

$result = mysql\_query("SELECT \* FROM CUSTOMERS

WHERE name = $matches[0]");

} else {

echo "user name not accepted";

}

To demonstrate the problem, consider this excerpt −

// supposed input

$name = "Qadir'; DELETE FROM CUSTOMERS;";

mysql\_query("SELECT \* FROM CUSTOMSRS WHERE name='{$name}'");

The function call is supposed to retrieve a record from the CUSTOMERS table where the name column matches the name specified by the user. Under normal circumstances, **$name** would only contain alphanumeric characters and perhaps spaces, such as the string ilia. But here, by appending an entirely new query to $name, the call to the database turns into disaster; the injected DELETE query removes all records from the CUSTOMERS table.

Fortunately, if you use MySQL, the **mysql\_query()** function does not permit query stacking or executing multiple SQL queries in a single function call. If you try to stack queries, the call fails.

However, other PHP database extensions, such as **SQLite** and **PostgreSQL** happily perform stacked queries, executing all the queries provided in one string and creating a serious security problem.

Preventing SQL Injection

You can handle all escape characters smartly in scripting languages like PERL and PHP. The MySQL extension for PHP provides the function **mysql\_real\_escape\_string()** to escape input characters that are special to MySQL.

if (get\_magic\_quotes\_gpc()) {

$name = stripslashes($name);

}

$name = mysql\_real\_escape\_string($name);

mysql\_query("SELECT \* FROM CUSTOMERS WHERE name='{$name}'");

The LIKE Quandary

To address the LIKE quandary, a custom escaping mechanism must convert user-supplied '%' and '\_' characters to literals. Use **addcslashes()**, a function that lets you specify a character range to escape.

$sub = addcslashes(mysql\_real\_escape\_string("%str"), "%\_");

// $sub == \%str\\_

mysql\_query("SELECT \* FROM messages

WHERE subject LIKE '{$sub}%'");

# SQL Interview Questions

What is the difference between SQL and MySQL or SQL Server?

SQL or Structured Query Language is a language; language that communicates with a relational database thus providing ways of manipulating and creating databases. MySQL and Microsoft’s SQL Server both are relational database management systems that use SQL as their standard relational database language.

What is the difference between SQL and PL/SQL?

PL/SQL is a dialect of SQL that adds procedural features of programming languages in SQL. It was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL.

What are various DDL commands in SQL? Give brief description of their purposes.

Following are various DDL or Data Definition Language commands in SQL −

* **CREATE −** it creates a new table, a view of a table, or other object in database.
* **ALTER −** it modifies an existing database object, such as a table.
* **DROP −** it deletes an entire table, a view of a table or other object in the database.

What are various DML commands in SQL? Give brief description of their purposes.

Following are various DML or Data Manipulation Language commands in SQL −

* **SELECT −** it retrieves certain records from one or more tables.
* **INSERT −** it creates a record.
* **UPDATE −** it modifies records.
* **DELETE −** it deletes records.

What are various DCL commands in SQL? Give brief description of their purposes.

Following are various DCL or Data Control Language commands in SQL −

* **GRANT −** it gives a privilege to user.
* **REVOKE −** it takes back privileges granted from user.

Can you sort a column using a column alias?

Yes. A column alias could be used in the ORDER BY clause.

Is a NULL value same as zero or a blank space? If not then what is the difference?

A NULL value is not same as zero or a blank space. A NULL value is a value which is ‘unavailable, unassigned, unknown or not applicable’. Whereas, zero is a number and blank space is a character.

Say True or False. Give explanation if False.

If a column value taking part in an arithmetic expression is NULL, then the result obtained would be NULLM.

True.

If a table contains duplicate rows, does a query result display the duplicate values by default? How can you eliminate duplicate rows from a query result?

A query result displays all rows including the duplicate rows. To eliminate duplicate rows in the result, the DISTINCT keyword is used in the SELECT clause.

What is the purpose of the condition operators BETWEEN and IN?

The BETWEEN operator displays rows based on a range of values. The IN condition operator checks for values contained in a specific set of values.

How do you search for a value in a database table when you don’t have the exact value to search for?

In such cases, the LIKE condition operator is used to select rows that match a character pattern. This is also called ‘wildcard’ search.

What is the default ordering of data using the ORDER BY clause? How could it be changed?

The default sorting order is ascending. It can be changed using the DESC keyword, after the column name in the ORDER BY clause.

What are the specific uses of SQL functions?

SQL functions have the following uses −

* Performing calculations on data
* Modifying individual data items
* Manipulating the output
* Formatting dates and numbers
* Converting data types

What are the case manipulation functions of SQL?

LOWER, UPPER, INITCAP

Which function returns the remainder in a division operation?

The MOD function returns the remainder in a division operation.

What is the purpose of the NVL function?

The NVL function converts a NULL value to an actual value.

What is the difference between the NVL and the NVL2 functions?

The NVL(exp1, exp2) function converts the source expression (or value) exp1 to the target expression (or value) exp2, if exp1 contains NULL. The return value has the same data type as that of exp1.

The NVL2(exp1, exp2, exp3) function checks the first expression exp1, if it is not null then, the second expression exp2 is returned. If the first expression exp1 is null, then the third expression exp3 is returned.

What is the use of the NULLIF function?

The NULLIF function compares two expressions. If they are equal, the function returns null. If they are not equal, the first expression is returned.

Discuss the ***Syntax*** and use of the COALESCE function?

The COALESCE function has the expression COALESCE(exp1, exp2, …. expn)

It returns the first non-null expression given in the parameter list.

Which expressions or functions allow you to implement conditional processing in a SQL statement?

There are two ways to implement conditional processing or IF-THEN-ELSE logic in a SQL statement.

* Using CASE expression
* Using the DECODE function

You want to display a result query from joining two tables with 20 and 10 rows respectively. Erroneously you forget to write the WHERE clause. What would be the result?

The result would be the Cartesian product of two tables with 20 x 10 = 200 rows.

What is the difference between cross joins and natural joins?

The cross join produces the cross product or Cartesian product of two tables. The natural join is based on all the columns having same name and data types in both the tables.

What is the purpose of the group functions in SQL? Give some ***Example***s of group functions.

Group functions in SQL work on sets of rows and returns one result per group. ***Example***s of group functions are AVG, COUNT, MAX, MIN, STDDEV, SUM, VARIANCE.

Say True or False. Give explanation if False.

By default the group functions consider only distinct values in the set.

By default, group functions consider all values including the duplicate values.

Say True or False. Give explanation if False.

The DISTINCT keyword allows a function consider only non-duplicate values.

True.

Say True or False. Give explanation if False.

All group functions ignore null values.

True.

Say True or False. Give explanation if False.

COUNT(\*) returns the number of columns in a table.

False. COUNT(\*) returns the number of rows in a table.

What’s wrong in the following query?

SELECT subject\_code, count(name)

FROM students;

It doesn’t have a GROUP BY clause. The subject\_code should be in the GROUP BY clause.

SELECT subject\_code, count(name)

FROM students

GROUP BY subject\_code;

What’s wrong in the following query?

SELECT subject\_code, AVG (marks)

FROM students

WHERE AVG(marks) > 75

GROUP BY subject\_code;

The WHERE clause cannot be used to restrict groups. The HAVING clause should be used.

SELECT subject\_code, AVG (marks)

FROM students

HAVING AVG(marks) > 75

GROUP BY subject\_code;

Say True or False. Give explanation if False.

Group functions cannot be nested.

False. Group functions can be nested to a depth of two.

What do you understand by a subquery? When is it used?

A subquery is a SELECT statement embedded in a clause of another SELECT statement. It is used when the inner query, or the subquery returns a value that is used by the outer query. It is very useful in selecting some rows in a table with a condition that depends on some data which is contained in the same table.

Say True or False. Give explanation if False.

A single row subquery returns only one row from the outer SELECT statement

False. A single row subquery returns only one row from the inner SELECT statement.

Say True or False. Give explanation if False.

A multiple row subquery returns more than one row from the inner SELECT statement.

True.

Say True or False. Give explanation if False.

Multiple column subqueries return more than one column from the inner SELECT statement.

True.

What’s wrong in the following query?

SELECT student\_code, name

FROM students

WHERE marks =

(SELECT MAX(marks)

FROM students

GROUP BY subject\_code);

Here a single row operator = is used with a multiple row subquery.

What are the various multiple row comparison operators in SQL?

IN, ANY, ALL.

What is the pupose of DML statements in SQL?

The DML statements are used to add new rows to a table, update or modify data in existing rows, or remove existing rows from a table.

Which statement is used to add a new row in a database table?

The INSERT INTO statement.

Say True or False. Give explanation if False.

While inserting new rows in a table you must list values in the default order of the columns.

True.

How do you insert null values in a column while inserting data?

Null values can be inserted into a table by one of the following ways −

* Implicitly by omitting the column from the column list.
* Explicitly by specifying the NULL keyword in the VALUES clause.

Say True or False. Give explanation if False.

INSERT statement does not allow copying rows from one table to another.

False. INSERT statement allows to add rows to a table copying rows from an existing table.

How do you copy rows from one table to another?

The INSERT statement can be used to add rows to a table by copying from another table. In this case, a subquery is used in the place of the VALUES clause.

What happens if you omit the WHERE clause in the UPDATE statement?

All the rows in the table are modified.

Can you modify the rows in a table based on values from another table? Explain.

Yes. Use of subqueries in UPDATE statements allow you to update rows in a table based on values from another table.

Say True or False. Give explanation if False.

The DELETE statement is used to delete a table from the database.

False. The DELETE statement is used for removing existing rows from a table.

What happens if you omit the WHERE clause in a delete statement?

All the rows in the table are deleted.

Can you remove rows from a table based on values from another table? Explain.

Yes, subqueries can be used to remove rows from a table based on values from another table.

Say True or False. Give explanation if False.

Attempting to delete a record with a value attached to an integrity constraint, returns an error.

True.

Say True or False. Give explanation if False.

You can use a subquery in an INSERT statement.

True.

What is the purpose of the MERGE statement in SQL?

The MERGE statement allows conditional update or insertion of data into a database table. It performs an UPDATE if the rows exists, or an INSERT if the row does not exist.

Say True or False. Give explanation if False.

A DDL statement or a DCL statement is automatically committed.

True.

What is the difference between VARCHAR2 AND CHAR datatypes?

VARCHAR2 represents variable length character data, whereas CHAR represents fixed length character data.

Say True or False. Give explanation if False.

A DROP TABLE statement can be rolled back.

False. A DROP TABLE statement cannot be rolled back.

Which SQL statement is used to add, modify or drop columns in a database table?

The ALTER TABLE statement.

What is a view? Why should you use a view?

A view is a logical snapshot based on a table or another view. It is used for −

* Restricting access to data;
* Making complex queries simple;
* Ensuring data independency;
* Providing different views of same data.

Say True or False. Give explanation if False.

A view doesn’t have data of its own.

True.