Various Syntax in SQL

All the examples 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;

An expression is a combination of one or more values, operators and SQL functions that evaluate to a value. These SQL EXPRESSIONs are like formulae and they are written in query language. You can also use them to query the database for a specific set of data.

### 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

Let us now discuss each of these in detail.

## Boolean Expressions

SQL Boolean Expressions fetch the data based on matching a single value. Following is the syntax −

SELECT column1, column2, columnN

FROM table\_name

WHERE SINGLE VALUE MATCHING EXPRESSION;

Consider the CUSTOMERS table having the following records −

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)

The following table is a simple example showing the usage of various SQL Boolean Expressions −

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. Following is the syntax −

SELECT numerical\_expression as OPERATION\_NAME

[FROM table\_name

WHERE CONDITION] ;

Here, the numerical\_expression is used for a mathematical expression or any formula. Following is a simple example showing the usage of SQL Numeric Expressions −

SQL> SELECT (15 + 6) AS ADDITION

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

| ADDITION |

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

| 21 |

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

1 row in set (0.00 sec)

There are several built-in functions like avg(), sum(), count(), etc., to perform what is known as the aggregate data calculations against a table or a specific table column.

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 is as shown below −

SQL> SELECT GETDATE();;

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

| GETDATE |

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

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

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

1 row in set (0.00 sec)

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

## Syntax

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

CREATE DATABASE DatabaseName;

Always the database name should be unique within the RDBMS.

## Example

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

SQL> CREATE DATABASE testDB;

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

SQL> SHOW DATABASES;

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

| Database |

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

| information\_schema |

| AMROOD |

| TUTORIALSPOINT |

| mysql |

| orig |

| test |

| testDB |

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

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

## Syntax

The basic syntax of DROP DATABASE statement is as follows −

DROP DATABASE DatabaseName;

Always the database name should be unique within the RDBMS.

## Example

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

SQL> DROP DATABASE testDB;

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

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

SQL> SHOW DATABASES;

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

| Database |

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

| information\_schema |

| AMROOD |

| TUTORIALSPOINT |

| mysql |

| orig |

| test |

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

6 rows in set (0.00 sec)

When you have multiple databases in your SQL Schema, then before starting your operation, you would need to select a database where all the operations would be performed.

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

## Syntax

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

USE DatabaseName;

Always the database name should be unique within the RDBMS.

## Example

You can check the available databases as shown below −

SQL> SHOW DATABASES;

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

| Database |

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

| information\_schema |

| AMROOD |

| TUTORIALSPOINT |

| mysql |

| orig |

| test |

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

6 rows in set (0.00 sec)

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

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

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

## Syntax

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

CREATE TABLE table\_name(

column1 datatype,

column2 datatype,

column3 datatype,

.....

columnN datatype,

PRIMARY KEY( one or more columns )

);

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

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

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

## Example

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

SQL> CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25) ,

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

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

SQL> DESC CUSTOMERS;

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

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

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

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

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

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

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

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

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

5 rows in set (0.00 sec)

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

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

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

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

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

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

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

### Example

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

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

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

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

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

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

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

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

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

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

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

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

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

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

INSERT INTO CUSTOMERS

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

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

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

## Populate one table using another table

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

Here is the syntax −

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

SELECT column1, column2, ...columnN

FROM second\_table\_name

[WHERE condition];

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

## Syntax

The basic syntax of the SELECT statement is as follows −

SELECT column1, column2, columnN FROM table\_name;

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

SELECT \* FROM table\_name;

## Example

Consider the CUSTOMERS table having the following records −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

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

SQL> SELECT ID, NAME, SALARY FROM CUSTOMERS;

This would produce the following result −

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

| ID | NAME | SALARY |

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

| 1 | Ramesh | 2000.00 |

| 2 | Khilan | 1500.00 |

| 3 | kaushik | 2000.00 |

| 4 | Chaitali | 6500.00 |

| 5 | Hardik | 8500.00 |

| 6 | Komal | 4500.00 |

| 7 | Muffy | 10000.00 |

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

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

SQL> SELECT \* FROM CUSTOMERS;

This would produce the result as shown below.

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

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

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

## Syntax

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

SELECT column1, column2, columnN

FROM table\_name

WHERE [condition]

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

## Example

Consider the CUSTOMERS table having the following records −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

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

SQL> SELECT ID, NAME, SALARY

FROM CUSTOMERS

WHERE SALARY > 2000;

This would produce the following result −

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

| ID | NAME | SALARY |

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

| 4 | Chaitali | 6500.00 |

| 5 | Hardik | 8500.00 |

| 6 | Komal | 4500.00 |

| 7 | Muffy | 10000.00 |

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

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

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

SQL> SELECT ID, NAME, SALARY

FROM CUSTOMERS

WHERE NAME = 'Hardik';

This would produce the following result −

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

| ID | NAME | SALARY |

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

| 5 | Hardik | 8500.00 |

The SQL **AND** & **OR** operators are used to combine multiple conditions to narrow data in an SQL statement. These two operators are called as the conjunctive operators.

These operators provide a means to make multiple comparisons with different operators in the same SQL statement.

## The AND Operator

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

### Syntax

The basic syntax of the AND operator with a WHERE clause is as follows −

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

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, which would 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;

This would produce the following 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

The basic syntax of the OR operator with a WHERE clause is as follows −

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

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 hasa query, which would 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;

This would produce the following result −

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

| ID | NAME | SALARY |

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

| 3 | kaushik | 2000.00 |

| 4 | Chaitali | 6500.00 |

| 5 | Hardik | 8500.00 |

| 6 | Komal | 4500.00 |

| 7 | Muffy | 10000.00 |

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

The SQL **UPDATE** Query is used to modify the existing records in a table. You can use the WHERE clause with the UPDATE query to update the selected rows, otherwise all the rows would be affected.

## Syntax

The basic syntax of the UPDATE query with a WHERE clause is as follows −

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

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 query will update the ADDRESS for a customer whose ID number is 6 in the table.

SQL> UPDATE CUSTOMERS

SET ADDRESS = 'Pune'

WHERE ID = 6;

Now, the CUSTOMERS table would have 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 | 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 UPDATE query would be enough as shown in the following code block.

SQL> UPDATE CUSTOMERS

SET ADDRESS = 'Pune', SALARY = 1000.00;

Now, CUSTOMERS table would have the following records −

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

| 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 |

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

The SQL DELETE Query is used to delete the existing records from a table.

You can use the WHERE clause with a DELETE query to delete the selected rows, otherwise all the records would be deleted.

## Syntax

The basic syntax of the DELETE query with the WHERE clause is as follows −

DELETE FROM table\_name

WHERE [condition];

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

## 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 has a query, which will DELETE a customer, whose ID is 6.

SQL> DELETE FROM CUSTOMERS

WHERE ID = 6;

Now, the CUSTOMERS table would have 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 |

| 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 would be as follows −

SQL> DELETE FROM CUSTOMERS;

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

The basic syntax of % and \_ 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 AND or OR operators. Here, XXXX could be any numeric or string value.

Example

The following table has a few examples showing 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. |

Let us take a real example, consider the CUSTOMERS table having the records as shown below.

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

Following is an example, which would 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 |

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

The SQL **TOP** clause is used to fetch a TOP N number or X percent records from a table.

**Note** − All the databases do not support the TOP clause. For example MySQL supports the **LIMIT** clause to fetch limited number of records while Oracle uses the **ROWNUM** command to fetch a limited number of records.

## Syntax

The basic syntax of the TOP clause with a SELECT statement would be as follows.

SELECT TOP number|percent column\_name(s)

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 |

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

The following query is an example on the SQL server, which would fetch the top 3 records from the CUSTOMERS table.

SQL> SELECT TOP 3 \* FROM CUSTOMERS;

This 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 |

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

If you are using MySQL server, then here is an equivalent example −

SQL> SELECT \* FROM CUSTOMERS

LIMIT 3;

This 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 |

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

If you are using an Oracle server, then the following code block has an equivalent example.

SQL> SELECT \* FROM CUSTOMERS

WHERE ROWNUM <= 3;

This 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 |

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

The SQL **ORDER BY** clause is used to sort the data in ascending or descending order, based on one or more columns. Some databases sort the query results in an ascending order by default.

## Syntax

The basic syntax of the ORDER BY clause is as follows −

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. Make sure whatever column you are using to sort that column should be in the column-list.

## 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 has an example, which would sort the result in an ascending order by the NAME and the SALARY −

SQL> SELECT \* FROM CUSTOMERS

ORDER BY NAME, SALARY;

This would produce the following 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 |

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

The following code block has an example, which would sort the result in the descending order by NAME.

SQL> SELECT \* FROM CUSTOMERS

ORDER BY NAME DESC;

This would produce the following 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 |

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

The SQL **GROUP BY** clause is used in collaboration with the SELECT statement to arrange identical data into groups. This GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.

## Syntax

The basic syntax of a GROUP BY clause is shown in the following code block. The GROUP BY clause must follow the conditions in the WHERE clause and must precede the ORDER BY clause if one is used.

SELECT column1, column2

FROM table\_name

WHERE [ conditions ]

GROUP BY column1, column2

ORDER BY column1, column2

## Example

Consider the CUSTOMERS table is 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 |

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

If you want to know the total amount of the salary on each customer, then the GROUP BY query would be as follows.

SQL> SELECT NAME, SUM(SALARY) FROM CUSTOMERS

GROUP BY NAME;

This would produce the following 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 |

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

Now, let us look at a table where the 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 |

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

Now again, if you want to know the total amount of salary on each customer, then the GROUP BY query would be as follows −

SQL> SELECT NAME, SUM(SALARY) FROM CUSTOMERS

GROUP BY NAME;

This would produce the following result −

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

| NAME | SUM(SALARY) |

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

| Hardik | 8500.00 |

| kaushik | 8500.00 |

| Komal | 4500.00 |

| Muffy | 10000.00 |

| Ramesh | 3500.00 |

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

The SQL **DISTINCT** keyword is used in conjunction with the SELECT statement to eliminate all the duplicate records and fetching only unique records.

There may be a situation when you have multiple duplicate records in a table. While fetching such records, it makes more sense to fetch only those unique records instead of fetching duplicate records.

## Syntax

The basic syntax of DISTINCT keyword to eliminate the 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 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 |

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

Now, let us use the DISTINCT keyword with the above SELECT query and then 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 |

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

The SQL **ORDER BY** clause is used to sort the data in ascending or descending order, based on one or more columns. Some databases sort the query results in an ascending order by default.

## Syntax

The basic syntax of the ORDER BY clause which would be used to sort the result in an ascending or descending order is as follows −

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. Make sure that whatever column you are using to sort, that column should be in the column-list.

## 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, which would sort the result in an ascending order by NAME and SALARY.

SQL> SELECT \* FROM CUSTOMERS

ORDER BY NAME, SALARY;

This would produce the following 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 |

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

The following code block has an example, which would sort the result in a descending order by NAME.

SQL> SELECT \* FROM CUSTOMERS

ORDER BY NAME DESC;

This would produce the following 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, the SELECT query used would be as follows −

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;

This would produce the following 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.

The SQL **Joins** clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each.

Consider the following two tables −

**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 |

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

Now, let us join these two tables in our SELECT statement as shown below.

SQL> SELECT ID, NAME, AGE, AMOUNT

FROM CUSTOMERS, ORDERS

WHERE CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

This would produce the following result.

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

| ID | NAME | AGE | AMOUNT |

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

| 3 | kaushik | 23 | 3000 |

| 3 | kaushik | 23 | 1500 |

| 2 | Khilan | 25 | 1560 |

| 4 | Chaitali | 25 | 2060 |

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

Here, it is noticeable 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. However, the most common operator is the 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 m

The SQL UNION clause/operator is used to combine the results of two or more SELECT statements without returning any duplicate rows.

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

But they need not have to be in the same length.

Syntax

The basic syntax of a **UNION** clause is as follows −

SELECT column1 [, column2 ]

FROM table1 [, table2 ]

[WHERE condition]

UNION

SELECT column1 [, column2 ]

FROM table1 [, table2 ]

[WHERE condition]

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

Example

Consider the following two tables.

**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 |

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

Now, let us join these two tables in our SELECT statement as follows −

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;

This would produce the following 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

The basic syntax of the **UNION ALL** is as follows.

SELECT column1 [, column2 ]

FROM table1 [, table2 ]

[WHERE condition]

UNION ALL

SELECT column1 [, column2 ]

FROM table1 [, table2 ]

[WHERE condition]

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

Example

Consider the following two tables,

**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 |

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

Now, let us join these two tables in our SELECT statement as follows −

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;

This would produce the following 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.
* he SQL **NULL** is the term used to represent a missing value. A NULL value in a table is a value in a field that appears to be blank.
* A field with a NULL value is a field with no value. It is very important to understand that a NULL value is different than a zero value or a field that contains spaces.

## Syntax

* The basic syntax of **NULL** while creating a table.
* SQL> CREATE TABLE CUSTOMERS(
* ID INT NOT NULL,
* NAME VARCHAR (20) NOT NULL,
* AGE INT NOT NULL,
* ADDRESS CHAR (25) ,
* SALARY DECIMAL (18, 2),
* PRIMARY KEY (ID)
* );
* Here, **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.
* Consider the following CUSTOMERS table having the records as shown below.
* +----+----------+-----+-----------+----------+
* | ID | NAME | AGE | ADDRESS | SALARY |
* +----+----------+-----+-----------+----------+
* | 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
* | 2 | Khilan | 25 | Delhi | 1500.00 |
* | 3 | kaushik | 23 | Kota | 2000.00 |
* | 4 | Chaitali | 25 | Mumbai | 6500.00 |
* | 5 | Hardik | 27 | Bhopal | 8500.00 |
* | 6 | Komal | 22 | MP | |
* | 7 | Muffy | 24 | Indore | |
* +----+----------+-----+-----------+----------+
* Now, following is the usage of the **IS NOT NULL**operator.
* SQL> SELECT ID, NAME, AGE, ADDRESS, SALARY
* FROM CUSTOMERS
* WHERE SALARY IS NOT NULL;
* This 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 |
* +----+----------+-----+-----------+----------+
* Now, following is the usage of the **IS NULL** operator.
* SQL> SELECT ID, NAME, AGE, ADDRESS, SALARY
* FROM CUSTOMERS
* WHERE SALARY IS NULL;
* This would produce the following result −
* +----+----------+-----+-----------+----------+
* | ID | NAME | AGE | ADDRESS | SALARY |
* +----+----------+-----+-----------+----------+
* | 6 | Komal | 22 | MP | |
* | 7 | Muffy | 24 | Indore | |
* +----+----------+-----+-----------+----------+

You can rename a table or a column temporarily by giving another name known as **Alias**. The use of table aliases is to rename a table in a specific SQL statement. The renaming is a temporary change and the actual table name does not change in the database. The column aliases are used to rename a table's columns for the purpose of a particular SQL query.

## Syntax

The basic syntax of a **table** alias is as follows.

SELECT column1, column2....

FROM table\_name AS alias\_name

WHERE [condition];

The basic syntax of a **column** alias is as follows.

SELECT column\_name AS alias\_name

FROM table\_name

WHERE [condition];

## Example

Consider the following two tables.

**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 |

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

Now, the following code block shows the 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;

This would produce the following result.

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

| ID | NAME | AGE | AMOUNT |

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

| 3 | kaushik | 23 | 3000 |

| 3 | kaushik | 23 | 1500 |

| 2 | Khilan | 25 | 1560 |

| 4 | Chaitali | 25 | 2060 |

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

Following is the usage of a **column alias**.

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

FROM CUSTOMERS

WHERE SALARY IS NOT NULL;

This would produce the following result.

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

| CUSTOMER\_ID | CUSTOMER\_NAME |

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

| 1 | Ramesh |

| 2 | Khilan |

| 3 | kaushik |

| 4 | Chaitali |

| 5 | Hardik |

| 6 | Komal |

| 7 | Muffy |

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

ndexes are **special lookup tables** that the database search engine can use to speed up data retrieval. Simply put, an index is a pointer to data in a table. An index in a database is very similar to an index in the back of a book.

For example, if you want to reference all pages in a book that discusses a certain topic, you first refer to the index, which lists all the topics alphabetically and are then referred to one or more specific page numbers.

An index helps to speed up **SELECT** queries and **WHERE** clauses, but it slows down data input, with the **UPDATE** and the **INSERT** statements. Indexes can be created or dropped with no effect on the data.

Creating an index involves the **CREATE INDEX** statement, which allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in an ascending or descending order.

Indexes can also be unique, like the **UNIQUE** constraint, in that the index prevents duplicate entries in the column or combination of columns on which there is an index.

## The CREATE INDEX Command

The basic syntax of a **CREATE INDEX** is as follows.

CREATE INDEX index\_name ON table\_name;

### Single-Column Indexes

A single-column index is created based on only one table column. The basic syntax is as follows.

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. The basic syntax is as follows.

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. Its basic syntax is as follows.

CREATE INDEX index\_name

on table\_name (column1, column2);

Whether to create a single-column index or a composite index, take into consideration the column(s) that you may use very frequently in a query's WHERE clause as filter conditions.

Should there be only one column used, a single-column index should be the choice. Should there be two or more columns that are frequently used in the WHERE clause as filters, the composite index would be the best choice.

### 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.

The basic syntax is as follows −

DROP INDEX index\_name;

You can check the [INDEX Constraint](https://www.tutorialspoint.com/sql/sql-index.htm) chapter to see some actual examples on Indexes.

### When should indexes be avoided?

Although indexes are intended to enhance a database's performance, there are times when they should be avoided.

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 SQL **ALTER TABLE** command is used to add, delete or modify columns in an existing table. You should also use the ALTER TABLE command to add and drop various constraints on an existing table.

## Syntax

* The basic syntax of an ALTER TABLE command to add a **New Column** in an existing table is as follows.
* ALTER TABLE table\_name ADD column\_name datatype;
* The basic syntax of an ALTER TABLE command to **DROP COLUMN** in an existing table is as follows.
* ALTER TABLE table\_name DROP COLUMN column\_name;
* The basic syntax of an ALTER TABLE command to change the **DATA TYPE** of a column in a table is as follows.
* ALTER TABLE table\_name MODIFY COLUMN column\_name datatype;
* The basic syntax of an ALTER TABLE command to add a **NOT NULL** constraint to a column in a table is as follows.
* ALTER TABLE table\_name MODIFY column\_name datatype NOT NULL;
* The basic syntax of ALTER TABLE to **ADD UNIQUE CONSTRAINT** to a table is as follows.
* ALTER TABLE table\_name
* ADD CONSTRAINT MyUniqueConstraint UNIQUE(column1, column2...);
* The basic syntax of an ALTER TABLE command to **ADD CHECK CONSTRAINT**to a table is as follows.
* ALTER TABLE table\_name
* ADD CONSTRAINT MyUniqueConstraint CHECK (CONDITION);
* The basic syntax of an ALTER TABLE command to **ADD PRIMARY KEY**constraint to a table is as follows.
* ALTER TABLE table\_name
* ADD CONSTRAINT MyPrimaryKey PRIMARY KEY (column1, column2...);
* The basic syntax of an ALTER TABLE command to **DROP CONSTRAINT** from a table is as follows.
* ALTER TABLE table\_name
* DROP CONSTRAINT MyUniqueConstraint;
* If you're using MySQL, the code is as follows −
* ALTER TABLE table\_name
* DROP INDEX MyUniqueConstraint;
* The basic syntax of an ALTER TABLE command to **DROP PRIMARY KEY**constraint from a table is as follows.
* ALTER TABLE table\_name
* DROP CONSTRAINT MyPrimaryKey;
* If you're using MySQL, the code is as follows −
* 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 |
* +----+----------+-----+-----------+----------+
* Following is the example to ADD a **New Column** to an existing table −
* ALTER TABLE CUSTOMERS ADD SEX char(1);
* Now, the 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 |
* +----+---------+-----+-----------+----------+------+
* Following is the example to DROP sex column from the existing table.
* ALTER TABLE CUSTOMERS DROP SEX;
* Now, the 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 |
* | 5 | Hardik | 27 | Bhopal | 8500.00 |
* | 6 | Komal | 22 | MP | 4500.00 |
* | 7 | Muffy | 24 | Indore | 10000.00 |
* +----+---------+-----+-----------+----------
* The SQL **TRUNCATE TABLE** command is used to delete complete data from an existing table.
* You can also use DROP TABLE command to delete complete table but it would remove complete table structure form the database and you would need to re-create this table once again if you wish you store some data.

## 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)

A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query.

A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create a view.

Views, which are a type of virtual tables allow users to do the following −

* Structure data in a way that users or classes of users find natural or intuitive.
* Restrict access to the data in such a way that a user can see and (sometimes) modify exactly what they need and no more.
* Summarize data from various tables which can be used to generate reports.

## Creating Views

Database views are created using the **CREATE VIEW** statement. Views can be created from a single table, multiple tables or another view.

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 in a similar way as you use them in a normal SQL SELECT query.

### 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.

The following code block has an example of creating same view 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.

So, if a view satisfies all the above-mentioned rules then you can update that view. The following code block has an 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. The same rules that apply to the UPDATE 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. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command.

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

Obviously, where you have a view, you need a way to drop the view if it is no longer needed. The syntax is very simple and is given below −

DROP VIEW view\_name;

Following is an example to drop the CUSTOMERS\_VIEW from the CUSTOMERS table.

DROP VIEW CUSTOMERS\_VIEW;

he **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

The HAVING clause must follow the GROUP BY clause in a query and must also precede the ORDER BY clause if used. The following code block has the 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 |

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

Following is an 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 |

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

### The COMMIT Command

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

The COMMIT command is the transactional command 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.

The 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 |

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

Following is an example which would 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 the transactional command 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.

The 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 |

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

Following is an example, which would 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.

The 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.

The syntax for rolling back to a SAVEPOINT is as shown below.

ROLLBACK TO SAVEPOINT\_NAME;

Following is an 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 ];

We have already discussed about the SQL LIKE operator, which is used to compare a value to similar values using the wildcard operators.

SQL supports two wildcard operators in conjunction with the LIKE operator which are explained in detail in the following table.

|  |  |
| --- | --- |
| **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. These symbols can be used in combinations.

## 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 following table has a number of examples showing 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. |

Let us take a real 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 is an example, which would 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 |

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

The following table has a list of all the important Date and Time related functions available through SQL. There are various other functions supported by your RDBMS. The given list is based on MySQL RDBMS.

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

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

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

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

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

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

| Saturday October 1997 |

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

1 row in set (0.00 sec)

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

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

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

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

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

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

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

1 row in set (0.00 sec)

DATE\_SUB(date,INTERVAL expr unit)

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

DAY(date)

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

DAYNAME(date)

Returns the name of the weekday for date.

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

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

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

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

| Thursday |

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

1 row in set (0.00 sec)

DAYOFMONTH(date)

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

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

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

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

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

| 3 |

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

1 row in set (0.00 sec)

DAYOFWEEK(date)

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

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

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

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

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

| 3 |

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

1 row in set (0.00 sec)

DAYOFYEAR(date)

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

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

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

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

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

| 34 |

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

1 row in set (0.00 sec)

EXTRACT(unit FROM date)

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

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

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

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

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

| 1999 |

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

1 row in set (0.00 sec)

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

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

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

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

| 199907 |

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

1 row in set (0.00 sec)

FROM\_DAYS(N)

Given a day number N, returns a DATE value.

mysql> SELECT FROM\_DAYS(729669);

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

| FROM\_DAYS(729669) |

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

| 1997-10-07 |

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

1 row in set (0.00 sec)

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

FROM\_UNIXTIME(unix\_timestamp)

FROM\_UNIXTIME(unix\_timestamp,format)

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

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

mysql> SELECT FROM\_UNIXTIME(875996580);

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

| FROM\_UNIXTIME(875996580) |

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

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

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

1 row in set (0.00 sec)

HOUR(time)

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

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

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

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

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

| 10 |

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

1 row in set (0.00 sec)

LAST\_DAY(date)

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

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

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

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

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

| 2003-02-28 |

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

1 row in set (0.00 sec)

LOCALTIME and LOCALTIME()

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

LOCALTIMESTAMP and LOCALTIMESTAMP()

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

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)

## 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]

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.

Subqueries are most frequently used with the SELECT statement. The basic syntax is as follows −

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.

The basic syntax is as follows.

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.

The basic syntax is as follows.

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 |

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

There may be a situation when you have multiple duplicate records in a table. While fetching such records, it makes more sense to fetch only unique records instead of fetching duplicate records.

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.

## 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 |

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