# STRUCTURE QUERY LANGUAGE

SQL is the standard language for Relational Database System. All the Relational Database Management Systems (RDMS) like MySQL, MS Access, Oracle, Sybase, Informix, Postgres, and SQL Server use SQL as their standard database language.
Also, they are using different dialects, such as −

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

## Schema:

A schema is **a collection of database objects like tables, triggers, stored procedures, etc**. A schema is connected with a user which is known as the schema owner. Database may have one or more schema. SQL Server have some built-in schema, for example: dbo, guest, sys, and INFORMATION_SCHEMA.

## Class:

A class is **a group of objects that share common properties and behavior.**
For example, we can consider a car as a class that has characteristics like steering wheels, seats, brakes, etc.

## Object:

An object is **a member (also called an instance) of a class.**

## SQL

SQL is a language to operate databases; it includes database creation, deletion, fetching rows, modifying rows, etc.

## Data Definition Language:

| S.No. | Command & Description |
|---|---|
| 1 | **CREATE**<br>Creates a new table, a view of a table, or other object in the database. |
| 2 | **ALTER**<br>Modifies an existing database object, such as a table. |
| 3 | **DROP**<br>Deletes an entire table, a view of a table or other objects in the database. |

## Data Manipulation Language:

| S.No. | Command & Description |
|---|---|
| **1** | **SELECT** |

| | |
|---|---|
| | Retrieves certain records from one or more tables. |
| 2 | **INSERT**<br>Creates a record. |
| 3 | **UPDATE**<br>Modifies records. |
| 4 | **DELETE**<br>Deletes records. |

**Data Control Language:**

| Sr.No. | Command & Description |
|---|---|
| 1 | **GRANT**<br>Gives a privilege to user. |
| 2 | **REVOKE**<br>Takes back privileges granted from user. |

The data in an RDBMS is stored in database objects which are called as **tables**.

**SQL Constraints**

Constraints are the rules enforced on data columns on a table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database.
Constraints could be either on a column level or a table level. The column level constraints are applied only to one column, whereas the table level constraints are applied to the whole table.

Following are some of the most used constraints available in SQL –

- NOT NULL Constraint − Ensures that a column cannot have a NULL value.
- DEFAULT Constraint − Provides a default value for a column when none is specified.
- UNIQUE Constraint − Ensures that all the values in a column are different.
- PRIMARY Key − Uniquely identifies each row/record in a database table. Primary key does not accept null value. Whereas Unique Key accept the null values.
- FOREIGN Key − Uniquely identifies a row/record in any another database table.

  **foreign key (table2_column_name) references table1 (table1_column_name)**

- CHECK Constraint − The CHECK constraint ensures that all values in a column satisfy certain conditions.
- INDEX − Used to create and retrieve data from the database very quickly.

**SQL Limit:**

Limit query is used to **restrict** the number of rows returns from the result set, rather than **fetching** the whole set in the MySQL database.

**Syntax**

```
SELECT column_list
FROM table_name
LIMIT offset, count;
```

**Offset:** It specifies the number of a row from which you want to return. The offset of the row starts from 0, not 1.
**Count:** It specifies the maximum number of rows you want to return.

**Various Syntax in SQL:**

| S.No | Query |
|---|---|
| **1** | **SQL SELECT Statement**<br>SELECT column1, column2....columnN<br>FROM   table_name; |
| **2** | **SQL DISTINCT Clause**<br>SELECT DISTINCT column1, column2....columnN<br>FROM   table_name; |
| **3** | **SQL WHERE Clause**<br>SELECT column1, column2....columnN<br>FROM   table_name<br>WHERE  CONDITION; |
| **4** | **SQL AND/OR Clause**<br>SELECT column1, column2....columnN<br>FROM   table_name<br>WHERE  CONDITION-1 {AND\|OR} CONDITION-2; |
| **5** | **SQL IN Clause**<br>SELECT column1, column2....columnN<br>FROM   table_name<br>WHERE  column_name IN (val-1, val-2,...val-N); |
| **6** | **SQL BETWEEN Clause**<br>SELECT column1, column2....columnN<br>FROM   table_name<br>WHERE  column_name BETWEEN val-1 AND val-2; |
| **7** | **SQL LIKE Clause**<br>SELECT column1, column2....columnN<br>FROM   table_name<br>WHERE  column_name LIKE { PATTERN }; |
| **8** | **SQL ORDER BY Clause**<br>SELECT column1, column2....columnN |

|  |  |
|---|---|
|  | FROM table_name<br>WHERE CONDITION<br>ORDER BY column_name {ASC\|DESC}; |
| **9** | **SQL GROUP BY Clause**<br>SELECT SUM(column_name)<br>FROM table_name<br>GROUP BY column_name; |
| **10** | **SQL COUNT Clause**<br>SELECT COUNT(column_name)<br>FROM table_name<br>WHERE CONDITION; |
| **11** | **SQL HAVING Clause**<br>SELECT SUM(column_name)<br>FROM table_name<br>GROUP BY column_name<br>HAVING (arithematic function condition); |
| **12** | **SQL CREATE TABLE Statement**<br>CREATE TABLE table_name(<br>column1 datatype,<br>column2 datatype,<br>column3 datatype,<br>.....<br>columnN datatype,<br>PRIMARY KEY( one or more columns )<br>); |
| **13** | **SQL DROP TABLE Statement**<br>DROP TABLE table_name; |
| **14** | **SQL CREATE INDEX Statement**<br>CREATE UNIQUE INDEX index_name<br>ON table_name ( column1, column2,...columnN); |
| **15** | **SQL DROP INDEX Statement**<br>ALTER TABLE table_name<br>DROP INDEX index_name; |
| **16** | **SQL DESC Statement**<br>DESC table_name; |
| **17** | **SQL TRUNCATE TABLE Statement**<br>TRUNCATE TABLE table_name; |
| **18** | **SQL ALTER TABLE Statement**<br>ALTER TABLE table_name {ADD\|DROP\|MODIFY} column_name {datatype};<br>SQL ALTER TABLE Statement (Rename) |

| | |
|---|---|
| | ALTER TABLE table_name RENAME TO new_table_name; |
| **19** | **SQL INSERT INTO Statement**<br>INSERT INTO table_name( column1, column2....columnN)<br>VALUES ( value1, value2....valueN); |
| **20** | **SQL UPDATE Statement**<br>UPDATE table_name<br>SET column1 = value1, column2 = value2....columnN=valueN<br>[ WHERE  CONDITION ]; |
| **21** | **SQL DELETE Statement**<br>DELETE FROM table_name<br>WHERE  {CONDITION}; |
| **22** | **SQL CREATE DATABASE Statement**<br>CREATE DATABASE database_name; |
| **23** | **SQL DROP DATABASE Statement**<br>DROP DATABASE database_name; |
| **24** | **SQL USE Statement**<br>USE database_name; |
| **25** | **SQL COMMIT Statement**<br>COMMIT; |
| **26** | **SQL ROLLBACK Statement**<br>ROLLBACK; |

**SQL Arithmetic Operator:**

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

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

**SQL Comparison Operator:**

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

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

**SQL Logical Operators:**

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

| S. No. | Operator & Description |
|---|---|
| **1** | ALL<br>The ALL operator is used to compare a value to all values in another value set. |
| **2** | AND<br>The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause. |
| **3** | ANY<br>The ANY operator is used to compare a value to any applicable value in the list as per the condition. |
| **4** | BETWEEN<br>The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value. |
| **5** | EXISTS<br>The EXISTS operator is used to search for the presence of a row in a specified table that meets a certain criterion. |
| **6** | IN |

| | |
|---|---|
| | The IN operator is used to compare a value to a list of literal values that have been specified. |
| **7** | LIKE<br>The LIKE operator is used to compare a value to similar values using wildcard operators. |
| **8** | NOT<br>The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. This is a negate operator. |
| **9** | OR<br>The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause. |
| **10** | IS NULL<br>The NULL operator is used to compare a value with a NULL value. |
| **11** | UNIQUE<br>The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates). |

**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)
```

We 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.
We 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.

**Like Clause:**

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

- The percent sign (%)
- The underscore (_)

**% (percent sign):** This wildcard character matches zero, one, or more than one character.
**(underscore sign):** This wildcard character matches only one or a single character.

**Syntax:**
**SELECT** column_Name1, column_Name2 ...., column_NameN **FROM** table_Name **WHERE** colu mn_name LIKE pattern;

**Order By Clause:**
The SQL **ORDER BY** clause is used to sort the data in ascending or descending order, based on one or more columns.

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

**Group By Clause:**
The **Group By** statement is used for organizing similar data into groups.

**The SELECT** statement is used with the **GROUP BY** clause in the SQL query.

**WHERE** clause is placed before the **GROUP BY** clause in **SQL**.

**ORDER BY** clause is placed after the **GROUP BY** clause in **SQL**.

**Syntax:**

**SELECT** column1, function_name(column2)

**FROM** table_name

**WHERE** condition

**GROUP BY** column1, column2

**ORDER BY** column1, column2;

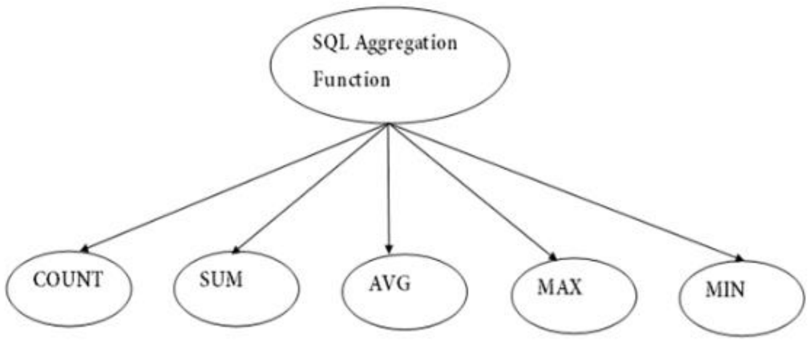
**Having Clause:**
The HAVING clause places the condition in the groups defined by the GROUP BY clause in the SELECT statement.

SELECT column_Name1, column_Name2, ....., column_NameN aggregate_function_name(colu mn_Name) FROM table_name GROUP BY column_Name1 HAVING condition;

WHERE Clause is used to filter the records from the table based on the specified condition. HAVING Clause is used to filter record from the groups based on the specified condition.

**Aggregate Function:**

SQL aggregation function is used to perform the calculations on multiple rows of a single column of a table. It returns a single value.
It is also used to summarize the data.



**Syntax:**

SELECT aggregate function(*)

FROM table_name

WHERE condition;

**Count Aggregate Function:**

It returns the total number of rows present in table, if query passed as **Count(*)**, it will returns total number of rows in table includes duplicates and null values. While if the query is passed as **Count(column name)**, it will returns total number of rows in column, includes duplicates value but null values is excluded.

And **Count(Distinct column name)** returns the total number of distinct rows in column, null values and duplicates values are excluded.

**Distinct Clause:**

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

```
SELECT DISTINCT column
FROM table_name
WHERE [condition]
```

**Distinct Clause with Multiple columns**

Distinct clause also works with multiple columns, but it will eliminates those rows where all the selected fields are identical.

```
SELECT DISTINCT column1, column2,.....columnN
FROM table_name
WHERE [condition]
```

Example:
We have a table called bugs, with the following data.

| id | course_id | exercise_id | reported_by |
|---|---|---|---|
| 1 | 5 | 4 | Tod |
| 2 | 5 | 4 | Alex |
| 3 | 5 | 3 | Roy |
| 4 | 5 | 4 | Roy |
| 5 | 7 | 4 | Alex |
| 6 | 7 | 8 | Tod |
| 7 | 14 | 2 | Alex |
| 8 | 14 | 4 | Tod |
| 9 | 14 | 6 | Tod |
| 10 | 14 | 2 | Roy |

```
SELECT DISTINCT course_id, exercise_id FROM bugs;
```

Output:

| course_id | exercise_id |
|---|---|
| 14 | 2 |
| 5 | 4 |
| 14 | 4 |
| 14 | 6 |
| 5 | 3 |
| 7 | 4 |
| 7 | 8 |

It is true that there are duplicated values in the course_id and exercise_id, but every row is unique (there are no two rows with the same value in course_id and exercise_id).

**Union Clause:**

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.

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]

UNION

SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

**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.

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]

UNION ALL

SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

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

- SQL INTERSECT Clause − 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 − This combines two SELECT statements and returns rows from the first SELECT statement that are not returned by the second SELECT statement.

**Alias Clause:**

It is used to rename a table or a column temporarily by giving another name known as **Alias.**

```
SELECT column1, column2....
FROM table_name AS alias_name
WHERE [condition];
```

**INTO Clause:**

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

SELECT INTO Syntax

Copy all columns into a new table:

```
SELECT *
INTO newtable
FROM oldtable
WHERE condition;
```

**SQL Case Statement:**

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

**Syntax:**

```
CASE
   WHEN condition1 THEN result1
   WHEN condition2 THEN result2
   WHEN conditionN THEN resultN
   ELSE result
END;
```

Example:
Below is a selection from the "OrderDetails" table in the Northwind sample database:

| OrderDetailID | OrderID | ProductID | Quantity |
|---|---|---|---|
| 1 | 10248 | 11 | 12 |
| 2 | 10248 | 42 | 10 |
| 3 | 10248 | 72 | 5 |
| 4 | 10249 | 14 | 9 |
| 5 | 10249 | 51 | 40 |

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

**Joins:**

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.
There are different types of joins available in SQL –

**INNER JOIN** – returns rows when there is a match in both tables.

```
SELECT table1.column1, table2.column2...
FROM table1
INNER JOIN table2
ON table1.common_field = table2.common_field;
```

**LEFT JOIN** – returns all rows from the left table, even if there are no matches in the right table.

```
SELECT table1.column1, table2.column2...
FROM table1
LEFT JOIN table2
ON table1.common_field = table2.common_field;
```

**RIGHT JOIN** – returns all rows from the right table, even if there are no matches in the left table.

```
SELECT table1.column1, table2.column2...
FROM table1
RIGHT JOIN table2
ON table1.common_field = table2.common_field;
```

**FULL JOIN** – returns all the rows from both joined tables, whether they have a matching row or not.

```
SELECT table1.column1, table2.column2...
FROM table1
FULL JOIN table2
ON table1.common_field = table2.common_field;
```

**SELF JOIN** – A self-join is a regular join, but the table is joined with itself.

```
SELECT a.column_name, b.column_name...
FROM table1 a, table1 b
WHERE a.common_field = b.common_field;
```

**CARTESIAN JOIN** – returns the Cartesian product of the sets of records from the two or more joined tables.

```
SELECT table1.column1, table2.column2...
FROM  table1, table2 [, table3 ]
```

**Indexing:**

Indexes are **special lookup tables** that the database search engine can use to speed up data retrieval.

```
CREATE INDEX index_name ON table_name;
```

**Truncate Command:**

TRUNCATE Command is a Data Definition Language operation. It is used to remove all the records from a table. It deletes all the records from an existing table **but not the table itself.** The structure or schema of the table is preserved.

```
TRUNCATE TABLE  table_name;
```

**Delete Command:**

The DELETE statement in SQL is a Data Manipulation Language(DML) Command. It is used to delete existing records from an existing table. We can delete a single record or multiple records depending on the condition specified in the query.

**Drop Command:**

DROP statement is a Data Definition Language(DDL) Command which is used to delete existing database objects. It can be used to delete databases, tables, views, triggers, etc.

**View:**

In SQL, a view is a virtual table based on the result-set of an SQL statement.
A view is nothing more than a SQL statement that is stored in the database with an associated name.

```
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE [condition];
```

**With Clause:**

The SQL WITH clause allows you to give a sub-query block a name (a process also called sub-query refactoring), which can be referenced in several places within the main SQL query.

**Syntax:**

```
WITH temporary_table_name as (
SQL Query 1
)
SQL Query 2
```

**Partition By:**

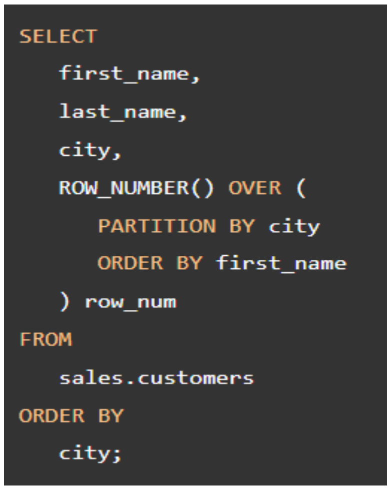
Row number generates the unique number for rows.
The OVER clause defines a window or user specified set of rows within a query result set.
Partition By Clause divides the query's result set into partition.

The Partition By clause divides the result set into the partition (another term a group of rows). The Row_Number () function is applied to each partition separately and reinitialized the row number for each partition. The Partition By clause is optional. If we skip it, Row_Number () function will treat the whole result set as a single partition. The Order By clauses define the logical order of the rows within each partition of result set. The Order By clause is mandatory because the Row_Number () function is order sensitive.

**Syntax:**

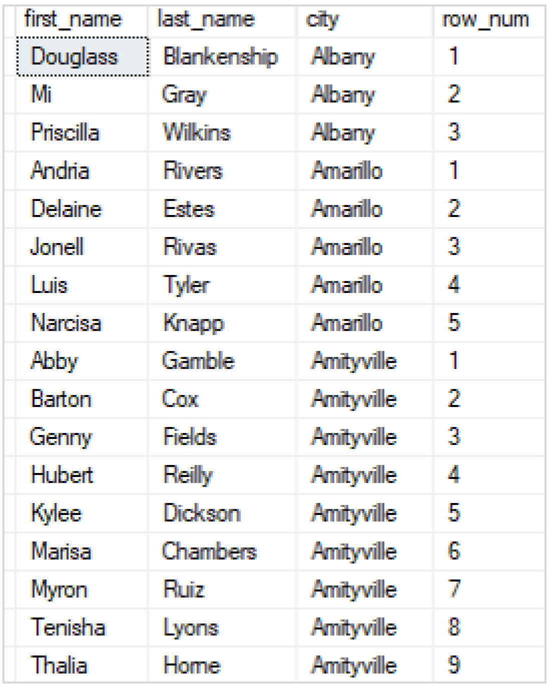
ROW_NUMBER() Over(Partition By Column_name Order By Column_name)

Example:



```
SELECT
   first_name,
   last_name,
   city,
   ROW_NUMBER() OVER (
      PARTITION BY city
      ORDER BY first_name
   ) row_num
FROM
   sales.customers
ORDER BY
   city;
```

The following picture shows the output.



| first_name | last_name | city | row_num |
|---|---|---|---|
| Douglass | Blankenship | Albany | 1 |
| Mi | Gray | Albany | 2 |
| Priscilla | Wilkins | Albany | 3 |
| Andria | Rivers | Amarillo | 1 |
| Delaine | Estes | Amarillo | 2 |
| Jonell | Rivas | Amarillo | 3 |
| Luis | Tyler | Amarillo | 4 |
| Narcisa | Knapp | Amarillo | 5 |
| Abby | Gamble | Amityville | 1 |
| Barton | Cox | Amityville | 2 |
| Genny | Fields | Amityville | 3 |
| Hubert | Reilly | Amityville | 4 |
| Kylee | Dickson | Amityville | 5 |
| Marisa | Chambers | Amityville | 6 |
| Myron | Ruiz | Amityville | 7 |
| Tenisha | Lyons | Amityville | 8 |
| Thalia | Horne | Amityville | 9 |

**Date Function:**

| Sr.No. | Function & Description |
|---|---|
| 1 | ADDDATE()<br>Adds dates |
| 2 | ADDTIME() |

| | |
|---|---|
| | Adds time |
| 3 | CONVERT_TZ() Converts from one timezone to another |
| 4 | CURDATE() Returns the current date |
| 5 | CURRENT_DATE(), CURRENT_DATE Synonyms for CURDATE() |
| 6 | CURRENT_TIME(), CURRENT_TIME Synonyms for CURTIME() |
| 7 | CURRENT_TIMESTAMP(), CURRENT_TIMESTAMP Synonyms for NOW() |
| 8 | CURTIME() Returns the current time |
| 9 | DATE_ADD() Adds two dates |
| 10 | DATE_FORMAT() Formats date as specified |
| 11 | DATE_SUB() Subtracts two dates |
| 12 | DATE() Extracts the date part of a date or datetime expression |
| 13 | DATEDIFF() Subtracts two dates |
| 14 | DAY() Synonym for DAYOFMONTH() |

| | |
|---|---|
| 15 | DAYNAME()<br>Returns the name of the weekday |
| 16 | DAYOFMONTH()<br>Returns the day of the month (1-31) |
| 17 | DAYOFWEEK()<br>Returns the weekday index of the argument |
| 18 | DAYOFYEAR()<br>Returns the day of the year (1-366) |
| 19 | EXTRACT<br>Extracts part of a date |
| 20 | FROM_DAYS()<br>Converts a day number to a date |
| 21 | FROM_UNIXTIME()<br>Formats date as a UNIX timestamp |
| 22 | HOUR()<br>Extracts the hour |
| 23 | LAST_DAY<br>Returns the last day of the month for the argument |
| 24 | LOCALTIME(), LOCALTIME<br>Synonym for NOW() |
| 25 | LOCALTIMESTAMP, LOCALTIMESTAMP()<br>Synonym for NOW() |
| 26 | MAKEDATE()<br>Creates a date from the year and day of year |
| 27 | MAKETIME<br>MAKETIME() |

| | |
|---|---|
| 28 | MICROSECOND()<br>Returns the microseconds from argument |
| 29 | MINUTE()<br>Returns the minute from the argument |
| 30 | MONTH()<br>Return the month from the date passed |
| 31 | MONTHNAME()<br>Returns the name of the month |
| 32 | NOW()<br>Returns the current date and time |
| 33 | PERIOD_ADD()<br>Adds a period to a year-month |
| 34 | PERIOD_DIFF()<br>Returns the number of months between periods |
| 35 | QUARTER()<br>Returns the quarter from a date argument |
| 36 | SEC_TO_TIME()<br>Converts seconds to 'HH:MM:SS' format |
| 37 | SECOND()<br>Returns the second (0-59) |
| 38 | STR_TO_DATE()<br>Converts a string to a date |
| 39 | SUBDATE()<br>When invoked with three arguments a synonym for DATE_SUB() |
| 40 | SUBTIME()<br>Subtracts times |

| | |
|---|---|
| 41 | SYSDATE()<br>Returns the time at which the function executes |
| 42 | TIME_FORMAT()<br>Formats as time |
| 43 | TIME_TO_SEC()<br>Returns the argument converted to seconds |
| 44 | TIME()<br>Extracts the time portion of the expression passed |
| 45 | TIMEDIFF()<br>Subtracts time |
| 46 | TIMESTAMP()<br>With a single argument this function returns the date or datetime expression. With two arguments, the sum of the arguments |
| 47 | TIMESTAMPADD()<br>Adds an interval to a datetime expression |
| 48 | TIMESTAMPDIFF()<br>Subtracts an interval from a datetime expression |
| 49 | TO_DAYS()<br>Returns the date argument converted to days |
| 50 | UNIX_TIMESTAMP()<br>Returns a UNIX timestamp |
| 51 | UTC_DATE()<br>Returns the current UTC date |
| 52 | UTC_TIME()<br>Returns the current UTC time |
| 53 | UTC_TIMESTAMP() |

| | |
|---|---|
| | Returns the current UTC date and time |
| 54 | WEEK()<br>Returns the week number |
| 55 | WEEKDAY()<br>Returns the weekday index |
| 56 | WEEKOFYEAR()<br>Returns the calendar week of the date (1-53) |
| 57 | YEAR()<br>Returns the year |
| 58 | YEARWEEK()<br>Returns the year and week |