SQL – Structured Query Language

# Introduction

## PhpMyAdmin

PhpMyAdmin is an open-source software tool introduced on **September 9**, **1998**, which is written in PHP. Basically, it is a third-party tool to manage the tables and data inside the database. PhpMyAdmin supports various type of operations on **MariaDB** and **MySQL**. The main purpose of phpMyAdmin is to handle the administration of MySQL over the web.

## communicating with RDBMS using SQLWhat is SQL?

SQL is Structured Query Language, which is a computer language for storing, manipulating, and retrieving data stored in a relational database. SQL is the standard language for Relational Database System. All the Relational Database Management Systems (RDMS) like MySQL, MS Access, Oracle, Sybase, Informix, Postgres, and SQL Server use SQL as their standard database language.

SQL is widely popular because it offers the following advantages –

* Allows users to access data in the relational database management systems.
* Allows users to describe the data.
* Allows users to define the data in a database and manipulate that data.
* Allows users to create and drop databases and tables.
* Allows users to set permissions on tables, procedures, and views.

## SQL Commands

The standard SQL commands to interact with relational databases are CREATE, SELECT, INSERT, UPDATE, DELETE and DROP. These commands can be classified into the following groups based on their nature –

### DDL - Data Definition Language

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

### DML - Data Manipulation Language

|  |  |
| --- | --- |
| Sr. No | Command & Description |
| 1 | SELECT  Retrieves certain records from one or more tables. |
| 2 | INSERT  Creates a record. |
| 3 | UPDATE  Modifies records. |
| 4 | DELETE  Deletes records. |

### DCL - Data Control Language

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

# SQL - RDBMS Concepts

RDBMS stands for **R**elational **D**atabase **M**anagement **S**ystem. RDBMS is the basis for SQL, and for all modern database systems like MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access. A Relational database management system (RDBMS) is a database management system (DBMS) that is based on the relational model as introduced by E. F. Codd.

* **What is a table?**

The data in an RDBMS is stored in database objects which are called as tables. This table is basically a collection of related data entries and it consists of numerous columns and rows. A table is the most common and simplest form of data storage in a relational database.

* **What is a field?**

Every table is broken up into smaller entities called fields. The fields in the CUSTOMERS table consist of ID, NAME, AGE, ADDRESS and SALARY. A field is a column in a table that is designed to maintain specific information about every record in the table.

* **What is a Record or a Row?**

A record is also called as a row of data is each individual entry that exists in a table.

* **What is a column?**

A column is a vertical entity in a table that contains all information associated with a specific field in a table.

# SQL – Operators

## What is an Operator in SQL?

An operator is a reserved word, or a character used primarily in an SQL statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations. These Operators are used to specify conditions in an SQL statement and to serve as conjunctions for multiple conditions in a statement.

### SQL Arithmetic Operators

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 Operators

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

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

# SQL - DATABASE

## SQL – CREATE DATABASE

* The SQL **CREATE DATABASE** statement is used to create a new SQL database. Always the database name should be unique within the RDBMS.
* **Syntax:** The basic syntax of this CREATE DATABASE statement is as follows:

CREATE DATABASE Database Name;

* **Example:** CREATE DATABASE demo;

## SQL - DROP or DELETE Database

* 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 Database Name;

* **Example**: DROP DATABASE demo;

## SQL - SELECT Database, USE Statement

* 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: USE Database Name;
* **Example**: SHOW DATABASES;

# SQL - Table

## SQL – CREATE TABLE

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. 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.

* **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));
* **Example:** CREATE TABLE customers(customer\_id int, customer\_name varchar(50), city varchar(50), grade int, salesman\_id int, credit int);

## Creating a Table from an Existing Table

* A copy of an existing table can be created using a combination of the CREATE TABLE statement and the SELECT statement. The new table has the same column definitions. All columns or specific columns can be selected. When you will create a new table using the existing table, the new table would be populated using the existing values in the old table.
* **Syntax:** Here, column1, column2... are the fields of the existing table and the same would be used to create fields of the new table. The basic syntax for creating a table from another table is as follows – CREATE TABLE NEW\_TABLE\_NAME AS SELECT [ column1, column2...columnN] FROM EXISTING\_TABLE\_NAME [ WHERE]
* **Example:** CREATE TABLE CUSTOMERS\_TEMP AS SELECT ID, SALARY FROM CUSTOMERS;

## SQL - Drop or Delete Table

* The SQL **DROP TABLE** statement is used to remove a table definition and all the data, indexes, triggers, constraints, and permission specifications for that table. You should be very careful while using this command because once a table is deleted then all the information available in that table will also be lost forever.
* **Syntax**: The basic syntax of this DROP TABLE statement is as follows:
* DROP TABLE table\_name;
* **Example:** DROP TABLE CUSTOMERS;

# SQL - Constraints

* Constraints are the rules enforced on the data columns of 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 commonly used constraints available in SQL.

## NOT NULL Constraint

* Ensures that a column cannot have NULL value. By default, a column can hold NULL values. If you do not want a column to have a NULL value, then you need to define such a constraint on this column specifying that NULL is now not allowed for that column. A NULL is not the same as no data, rather, it represents unknown data.
* **Example:**
* The following SQL query creates a new table called CUSTOMERS and adds 6 columns, out of which are customer\_id, customer\_name in this we specify not to accept NULLs –

CREATE TABLE CUSTOMERS (customer\_id INT NOT NULL, customer\_name VARCHAR (20) NOT NULL, city VARCHAR (25), grade int, salesman\_id int, credit int, PRIMARY KEY (customer\_id));

* If CUSTOMERS table has already been created, then to add a NOT NULL constraint to the credit column in MySQL, you would write a query like the one that is shown in the following:

ALTER TABLE CUSTOMERS MODIFY credit int NOT NULL;

## DEFAULT Constraint

The DEFAULT constraint provides a default value to a column when the INSERT INTO statement does not provide a specific value.

* **Example:**
* For example, the following SQL creates a new table called CUSTOMERS and adds five columns. Here, the SALARY column is set to 5000.00 by default, so in case the INSERT INTO statement does not provide a value for this column, then by default this column would be set to 5000.00:

CREATE TABLE CUSTOMERS (ID INT NOT NULL, NAME VARCHAR (50) NOT NULL, AGE INT NOT NULL, ADDRESS VARCHAR (25), SALARY DECIMAL (18, 2) DEFAULT 5000.00, PRIMARY KEY (ID));

* If the CUSTOMERS table has already been created, then to add a DEFAULT constraint to the SALARY column, you would write a query like the one which is shown in the code block below.

ALTER TABLE CUSTOMERS MODIFY SALARY DECIMAL (18, 2) DEFAULT 5000.00;

* **Drop Default Constraint:** To drop a DEFAULT constraint, use the following SQL query:

ALTER TABLE CUSTOMERS ALTER COLUMN SALARY DROP DEFAULT;

## [UNIQUE Constraint](https://www.tutorialspoint.com/sql/sql-unique.htm)

The UNIQUE Constraint prevents two records from having identical values in a column. In the CUSTOMERS table, for example, you might want to prevent two or more people from having an identical age.

* Example:
* For example, the following SQL query creates a new table called CUSTOMERS and adds five columns. Here, the AGE column is set to UNIQUE, so that you cannot have two records with the same age:

CREATE TABLE CUSTOMERS (ID INT NOT NULL, NAME VARCHAR (20) NOT NULL, AGE INT NOT NULL UNIQUE, ADDRESS CHAR (25), SALARY DECIMAL (18, 2), PRIMARY KEY (ID));

* If the CUSTOMERS table has already been created, then to add a UNIQUE constraint to the AGE column. You would write a statement like the query that is given in the code block below.

ALTER TABLE CUSTOMER MODIFY AGE INT NOT NULL UNIQUE;

* You can also use the following syntax, which supports naming the constraint in multiple columns as well.

ALTER TABLE CUSTOMERS ADD CONSTRAINT myUniqueConstraint UNIQUE (AGE, SALARY);

* If you are using MySQL, then you can use the following syntax – ALTER TABLE CUSTOMERS DROP INDEX myUniqueConstraint;

## [PRIMARY Key](https://www.tutorialspoint.com/sql/sql-primary-key.htm) Constraint

A primary key is a field in a table which uniquely identifies each row/record in a database table. Primary keys must contain unique values. A primary key column cannot have NULL values. A table can have only one primary key, which may consist of single or multiple fields. When multiple fields are used as a primary key, they are called a composite key. If a table has a primary key defined on any field(s), then you cannot have two records having the same value of that field(s).

* Example:

1. Here is the syntax to define the ID attribute as a primary key in a CUSTOMERS table: CREATE TABLE CUSTOMERS (ID INT NOT NULL, NAME VARCHAR (20) NOT NULL, AGE INT NOT NULL, ADDRESS VARCHAR (25), SALARY DECIMAL (18, 2), PRIMARY KEY (ID));
2. To create a PRIMARY KEY constraint on the "ID" column when the CUSTOMERS table already exists, use the following SQL syntax : ALTER TABLE CUSTOMER ADD PRIMARY KEY (ID);

* **Delete Primary Key**: You can clear the primary key constraints from the table with the syntax given below : ALTER TABLE CUSTOMERS DROP PRIMARY KEY;

## [FOREIGN Key](https://www.tutorialspoint.com/sql/sql-foreign-key.htm) Constraint

A foreign key is a key used to link two tables together. This is sometimes also called as a referencing key. A Foreign Key is a column or a combination of columns whose values match a Primary Key in a different table. **The relationship between 2 tables matches the Primary Key in one of the tables with a Foreign Key in the second table.** If a table has a primary key defined on any field(s), then you cannot have two records having the same value of that field(s).

* Example: Consider the structure of the following two tables.
* **CUSTOMERS table:** CREATE TABLE CUSTOMERS (ID INT NOT NULL, NAME VARCHAR (20) NOT NULL, AGE INT NOT NULL, ADDRESS VARCHAR (25), SALARY DECIMAL (18, 2), PRIMARY KEY (ID));
* **ORDERS table**: CREATE TABLE ORDERS (ID INT NOT NULL, DATE DATETIME, CUSTOMER\_ID INT references CUSTOMERS(ID), AMOUNT double, PRIMARY KEY (ID));

If the ORDERS table has already been created and the foreign key has not yet been set, the use the syntax for specifying a foreign key by altering a table: ALTER TABLE ORDERS: ADD FOREIGN KEY (Customer\_ID) REFERENCES CUSTOMERS (ID);

* **DROP a FOREIGN KEY Constraint**:To drop a FOREIGN KEY constraint, use the following SQL syntax: ALTER TABLE ORDERS DROP FOREIGN KEY;

## [CHECK Key Constraint](https://www.tutorialspoint.com/sql/sql-check.htm)

The CHECK constraint ensures that all the values in a column satisfies certain conditions. The CHECK Constraint enables a condition to check the value being entered into a record. If the condition evaluates to false, the record violates the constraint and is not entered the table.

* Example:
* For example, the following program creates a new table called CUSTOMERS and adds five columns. Here, we add a CHECK with AGE column, so that you cannot have any CUSTOMER who is below 18 years: CREATE TABLE CUSTOMERS (ID INT NOT NULL, NAME VARCHAR (20) NOT NULL, AGE INT NOT NULL CHECK (AGE >= 18), ADDRESS CHAR (25), SALARY DECIMAL (18, 2), PRIMARY KEY (ID));
* If the CUSTOMERS table has already been created, then to add a CHECK constraint to AGE column, you would write a statement like the one given below: ALTER TABLE CUSTOMERS MODIFY AGE INT NOT NULL CHECK (AGE >= 18);
* You can also use the following syntax, which supports naming the constraint in multiple columns as well – ALTER TABLE CUSTOMERS ADD CONSTRAINT myCheckConstraint CHECK (AGE >= 18);
* **To drop a CHECK constraint,** use the following SQL syntax. This syntax does not work with MySQL: ALTER TABLE CUSTOMERS DROP CONSTRAINT myCheckConstraint;

# SQL – Insert Query

* The SQL **INSERT INTO** Statement is used to add new rows of data to a table in the database.
* **Syntax:** There are two basic syntaxes of the INSERT INTO statement which are shown below. INSERT INTO TABLE\_NAME (column1, column2, column3,...columnN) VALUES (value1, value2, value3,...valueN);
* Here, column1, column2, column3, columnN are the names of the columns in the table into which you want -to insert the data. You may not need to specify the column(s) name in the SQL query if you are adding values for all the columns of the table. But make sure the order of the values is in the same order as the columns in the table.
* The **SQL INSERT INTO** syntax will be as follows – INSERT INTO TABLE\_NAME VALUES (value1, value2, value3, valueN);
* **Example:**  INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY) VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00);

# SQL – Select Query

* 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.

* **Example:**  Select id, name, age, address from customers;
* **Syntax:** If you want to fetch all the fields available in the field, then you can use the following syntax:

SELECT \* FROM table\_name;

* **Example:** Select \* from customers;

# SQL - Distinct Keyword

* 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:** SELECT DISTINCT SALARY FROM CUSTOMERS;

# SQL - WHERE Clause

* The SQL **WHERE** clause is used to specify a condition while fetching the data from a single table or by joining with multiple tables. If the given condition is satisfied, then only it returns a specific value from the table. You should use the WHERE clause to filter the records and fetching only the necessary records. The WHERE clause is not only used in the SELECT statement, but it is also used in the UPDATE, DELETE statement, etc.,
* **Syntax:** The basic syntax of the SELECT statement with the WHERE clause is as shown: SELECT column1, column2, columnN FROM table\_name WHERE [condition]

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.

* **Example**: SELECT ID, NAME, SALARY FROM CUSTOMERS WHERE NAME = 'Hardik';

# SQL – Update Query

* 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];
* **Example**: UPDATE CUSTOMERS SET ADDRESS = 'Pune' WHERE ID = 6;

UPDATE CUSTOMERS SET ADDRESS = 'Pune', SALARY = 1000.00;

# SQL – Delete Query

* 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. You can combine N number of conditions using AND or OR operators.
* **Syntax:** The basic syntax of the DELETE query with the WHERE clause is as follows − DELETE FROM table\_name WHERE [condition];
* **Example:** DELETE FROM CUSTOMERS WHERE ID = 6;

# SQL – AND/OR Conjunctive Operators

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:** SELECT ID, NAME, SALARY FROM CUSTOMERS WHERE SALARY > 2000 AND age < 25;

## 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:** SELECT ID, NAME, SALARY FROM CUSTOMERS WHERE SALARY > 2000 OR age < 25;

# SQL - BETWEEN OPERATORS

* The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates. The BETWEEN operator is inclusive: begin and end values are included.
* **Syntax:** SELECT column\_name(s) FROM table\_name WHERE column\_name BETWEEN value1 AND value2;
* **Example:** SELECT \* FROM Products WHERE Price BETWEEN 10 AND 20;

# SQL - IN Operator

* The IN operator allows you to specify multiple values in a WHERE clause. The IN operator is a shorthand for multiple OR conditions.
* **Syntax:** SELECT column\_name(s) FROM table\_name WHERE column\_name IN (value1, value2, ...);
* **Example:** The following SQL statement selects all customers that are in "Germany", "France" or "UK":

SELECT \* FROM Customers WHERE Country IN ('Germany', 'France', 'UK');

The following SQL statement selects all customers that are NOT located in "Germany", "France" or "UK":

SELECT \* FROM Customers WHERE Country NOT IN ('Germany', 'France', 'UK');

# SQL - LIKE Clause

* The SQL **LIKE** clause is used to compare a value to similar values using wildcard operators. There are two wildcards used in conjunction with the LIKE operator. The percent sign (%) and 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%'

SELECT FROM table\_name WHERE column LIKE '%XXXX%'

SELECT FROM table\_name WHERE column LIKE 'XXXX\_'

SELECT FROM table\_name WHERE column LIKE '\_XXXX'

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. |

# SQL - ORDER BY Clause

* The SQL **ORDER BY** clause is used to sort the data in ascending or descending order, based on one column. 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];

* **Example:**

SELECT \* FROM customer ORDER BY customer\_id;

# SQL – Group by Clause

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

SELECT NAME, SUM (SALARY) FROM CUSTOMERS GROUP BY NAME;

# SQL – ALL and ANY

* **ALL & ANY** are logical operators in SQL. They return Boolean value as a result.
* **ALL**
* ALL operator is used to select all tuples of SELECT STATEMENT. It is also used to compare a value to every value in another value set or result from a subquery.
* The ALL operator returns TRUE if all of the subquery’s values meet the condition. The ALL must be preceded by comparison operators and evaluates true if all of the subquery’s values meet the condition.
* ALL is used with SELECT, WHERE, HAVING statement.
* **ALL with select statement Syntax:** SELECT ALL fieldnamesFROM table\_nameWHERE condition(s);
* **ALL with WHERE or HAVING Statement Syntax:** SELECT column\_name(s) FROM table\_name WHERE column\_name comparison operator ALL (SELECT column\_name FROM table\_name WHERE condition(s));
* **ANY**
* ANY compares a value to each value in a list or results from a query and evaluates to true if the result of an inner query contains at least one row.
* ANY return true if any of the subquery’s values meet the condition.
* ANY must be preceded by comparison operators.
* **Syntax:** SELECT column\_name(s) FROM table\_name WHERE column\_name comparison operator ANY (SELECT column\_name FROM table\_name WHERE condition(s));
* **EXAMPLE:** [select](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) \* from customer WHERE salesman\_id = ANY([SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) salesman\_id from salesman WHERE salesman\_city="Paris");

# SQL Functions

## SQL Server Aggregate Functions

An aggregate function performs a calculation one or more values and returns a single value. The aggregate function is often used with the [GROUP BY](http://www.sqlservertutorial.net/sql-server-basics/sql-server-group-by/) clause and [HAVING](http://www.sqlservertutorial.net/sql-server-basics/sql-server-having/) clause of the [SELECT](http://www.sqlservertutorial.net/sql-server-basics/sql-server-select/) statement.

### AVG () function

* AVG () function retrieves the average value of a given expression. If the function does not find a matching row, it returns NULL.
* **Syntax:** AVG ([DISTINCT] expr): Where expr is a given expression. The DISTINCT option can be used to return the average of the distinct values of expr.
* **Example:**
* **AVG () function:** SELECT AVG (credit) FROM customer;
* **AVG () function with group by:** SELECT grade, AVG (customer\_id) FROM customer GROUP BY salesman\_id;
* **AVG () function with distinct:** SELECT AVG (DISTINCT (grade)) FROM purchase;
* **AVG () function with COUNT () function:** SELECT pub\_id, COUNT (customer\_id), AVG (credit) FROM customer GROUP BY salesman\_id;
* **AVG () function with having:** SELECT customer\_id, AVG (credit) FROM customer GROUP BY salesman\_id HAVING salesman\_id =5001;

### COUNT () function

* COUNT () function returns a count of a number of non-NULL values of a given expression.
* **Example:**
* **COUNT () function:** SELECT COUNT (credit) FROM customer
* **COUNT () function with Logical Operator:** SELECT city, COUNT (\*) FROM customer WHERE city='New York' OR city='London' GROUP BY city;

### MIN () function

* MIN () function returns the minimum value of an expression. MIN () function returns NULL when the return set has no rows.
* **Example:**
* **MIN () function**: SELECT MIN (credit) FROM customer;
* **MIN () function with group by**: SELECT order\_no, MIN (purchase\_amt) FROM orders GROUP BY customer\_id;
* **MIN() function with group by and order by** [SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) order\_no, [MIN](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/group-by-functions.html#function_min)(purchase\_amt) FROM orders GROUP BY customer\_id ORDER BY order\_no;
* **MIN() function with having** : [SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) order\_no, [MIN](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/group-by-functions.html#function_min)(purchase\_amt) FROM orders GROUP BY customer\_id HAVING [MIN](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/group-by-functions.html#function_min)(purchase\_amt)>2000 ORDER BY order\_no;
* **MIN() function with distinct** : [SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) [MIN](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/group-by-functions.html#function_min)(DISTINCT purchase\_amt) FROM orders GROUP BY customer\_id;

### MAX () function

* MAX () function: MAX () function returns the maximum value of an expression.
* **Examples:**
* **MAX() function with group by** : [SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) order\_no, [MAX](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/group-by-functions.html#function_max)(purchase\_amt) FROM orders GROUP BY customer\_id;
* **MAX with group by and order by**: [SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) order\_no, [MAX](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/group-by-functions.html#function_max)(purchase\_amt) FROM orders GROUP BY customer\_id ORDER BY order\_no;
* **MAX () function with having**:

MySQL MAX() function retrieves the maximum value from an expression which has undergone a grouping operation by GROUP BY clause and filtered using HAVING clause followed by some condition : [SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) city,[MAX](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/group-by-functions.html#function_max)(credit) FROM customer GROUP BY salesman\_id HAVING [MAX](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/group-by-functions.html#function_max)(credit)>=20000;

### SUM () function

* MySQL SUM () function returns the sum of an expression. SUM () function returns NULL when the return set has no rows.
* **Example**: SELECT City, COUNT(Cust\_Id) as 'No of customers', sum(credit) as 'Sum of Credit Amount', MAX (credit) as 'Maximum Credit Amount', MIN (credit) as 'Minimum Credit Amount', AVG (credit) as 'Average credit amount' from customer GROUP BY City;
* [select](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) grade, [avg](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/group-by-functions.html#function_avg) (credit) 'Avg Amount', [max](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/group-by-functions.html#function_max)(credit) as 'Maximum credit’, min(credit) as 'Minimum Credit' from customer GROUP by(grade);
* [select](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) grade, [avg](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/group-by-functions.html#function_avg)(credit) as 'Avg Amount', [max](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/group-by-functions.html#function_max)(credit) as 'Maximum credit', [min](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/group-by-functions.html#function_min)(credit) as 'Minimum Credit' from customer GROUP by grade having [AVG](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/group-by-functions.html#function_avg)(credit)>=10000 ORDER by grade DESC;

## SQL String Functions

### CONCAT Function

* The MySQL CONCAT function takes one or more string arguments and concatenates them into a single string. The CONCAT function requires a minimum of one parameter otherwise it raises an error. The following illustrates the syntax of the CONCAT function.
* **Syntax:** CONCAT (string1, string2, ...);
* **Example:** [SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) concat(Cust\_Name,' ',city) NameCity FROM customer;

### LOWER Function

* The LOWER () function accepts a string argument and returns the lowercase version of that string.
* **Syntax:** LOWER (str) or LCASE (str)
* **Example:**

[SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) LOWER (Cust\_Name), LCASE (City) FROM `customer`;

[SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) Cust\_Name, LOWER (Cust\_Name) lowercase FROM customer ORDER BY Cust\_Name LIMIT 10;

### UPPER Function

* The UPPER () function returns the uppercase of a specified string argument. The following shows the syntax of the UPPER () function:
* **Syntax:** UPPER (str) or UCASE (str)

## Date Functions

### CURDATE Function

* The CURDATE () function returns the current [date](https://www.mysqltutorial.org/mysql-date/) as a value in the 'YYYY-MM-DD' format if it is used in a string context or YYYYMMDD format if it is used in a numeric context. The following example shows how the CURDATE () function is used in the string context.
* **Example**:
* The following example shows how the CURDATE () function is used in the string context: SELECT CURDATE ();
* And the following example illustrates how the CURDATE() function is used in a numeric context: [SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) CURDATE() + 0;
* The CURRENT\_DATE and CURRENT\_DATE() are synonyms for CURDATE() : [SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) [CURRENT\_DATE](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/date-and-time-functions.html%23function_current_date)(), [CURRENT\_DATE](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/date-and-time-functions.html%23function_current_date), CURDATE();
* CURDATE vs. NOW: The CURDATE () function returns the current date with date part only while the [NOW ()](https://www.mysqltutorial.org/mysql-now/) function returns both date and time parts of the current time. The result of the CURDATE () function is equivalent to the following expression: SELECT DATE (NOW ());

### DATEDIFF Function

* The MySQL DATEDIFF function calculates the number of days between two [DATE](https://www.mysqltutorial.org/mysql-date/), [DATETIME](https://www.mysqltutorial.org/mysql-datetime/), or [TIMESTAMP](https://www.mysqltutorial.org/mysql-timestamp.aspx) values.
* The syntax of the MySQL DATEDIFF function is as follows: DATEDIFF (date\_expression\_1, date\_expression\_2);
* The DATEDIFF function accepts two arguments that can be any valid date or date-time values. If you pass DATETIME or TIMESTAMP values, the DATEDIFF function only takes the date parts for calculation and ignores the time parts. The DATEDIFF function is useful in many cases e.g., you can calculate an [interval](https://www.mysqltutorial.org/mysql-interval/) in days that the products need to ship to a customer.
* **Example:**
* [SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) DATEDIFF ('2011-08-17','2011-08-17');
* [SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) DATEDIFF ('2011-08-17','2011-08-08');
* To calculate the number of days between the required date and shipped date of the orders, you use the DATEDIFF function as follows:

SELECT orderNumber, DATEDIFF (requiredDate, shippedDate) daysLeft FROM orders ORDER BY daysLeft DESC;

* The following statement gets all orders whose statuses are in-process and calculates the number of days between ordered date and required date: SELECT orderNumber, DATEDIFF (requiredDate, orderDate) remaining\_days FROM orders WHERE status = 'In Process' ORDER BY remaining\_days;

### DAY Function

* The DAY () function returns the day of the month of a given [date](https://www.mysqltutorial.org/mysql-date/). The following shows the syntax of the DAY function: DAY (date);
* The DAY () function accepts one argument that is a date value for which you want to get the day of the month. If the date argument is zero e.g., '0000-00-00', the DAY () function returns 0. In case the date is [NULL](https://www.mysqltutorial.org/mysql-null/), the DAY () function returns NULL. Note that DAY () function is the synonym of the DAYOFMONTH () function.
* **Example:**
* MySQL DAY() function simple example : [SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) DAY('2010-01-15');
* Using MySQL DAY() function to get the number of days in a month of a date: [SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) DAY(LAST\_DAY('2016-02-03'));
* Using MySQL DAY () function with a table example: SELECT DAY (orderDate) dayofmonth, COUNT (\*) FROM orders WHERE YEAR (orderDate) = 2004 GROUP BY dayofmonth ORDER BY dayofmonth;

### DATE\_ADD Function

* The DATE\_ADD function adds an [interval](https://www.mysqltutorial.org/mysql-interval/) to a [DATE](https://www.mysqltutorial.org/mysql-date/) or [DATETIME](https://www.mysqltutorial.org/mysql-datetime/) value. The following illustrates the syntax of the DATE\_ADD function: DATE\_ADD(start\_date, INTERVAL expr unit);
* The DATE\_ADD function takes two arguments:
* start\_date is a starting DATE or DATETIME value
* INTERVAL expr unit is an interval value to be added to the starting date value.
* The DATE\_ADD function may return a DATETIME value or a string, depending on the arguments:
* DATETIME if the first argument is a DATETIME value or if the interval value has time element such as hour, minute or second, etc.
* String otherwise.
* **Examples:**
* Add 1 second to 1999-12-31 23:59:59: [SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) DATE\_ADD('1999-12-31 23:59:59', [INTERVAL](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/comparison-operators.html%23function_interval) 1 SECOND) result;
* Add 1 day to 1999-12-31 00:00:01: [SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) DATE\_ADD('1999-12-31 00:00:01', [INTERVAL](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/comparison-operators.html%23function_interval) 1 DAY) result;
* Add 1 minute and 1 second to 1999-12-31 23:59:59 : [SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) DATE\_ADD('1999-12-31 23:59:59', [INTERVAL](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/comparison-operators.html%23function_interval) '1:1' MINUTE\_SECOND) result;
* Add -1 day and 5 hours to 2000-01-01 00:00:00 : [SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) DATE\_ADD('2000-01-01 00:00:00', [INTERVAL](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/comparison-operators.html%23function_interval) '-1 5' DAY\_HOUR) result;
* Add 1 second and 999999 microseconds to 1999-12-31 23:59:59.000002: SELECT DATE\_ADD ('1999-12-31 23:59:59.000002’, INTERVAL '1.999999' SECOND\_MICROSECOND) result;

### DATE\_FORMAT Function

* To format a [date](https://www.mysqltutorial.org/mysql-date/) value to a specific format, you use the DATE\_FORMAT function. The syntax of the DATE\_FORMAT function is as follows:

DATE\_FORMAT (date, format)

* The DATE\_FORMAT function accepts two arguments:
* date: is a valid date value that you want to format
* Format: is a format string that consists of predefined specifiers. Each specifier is preceded by a percentage character ( % ). See the table below for a list of predefined specifiers.
* The DATE\_FORMAT function returns a string whose [character set](https://www.mysqltutorial.org/mysql-character-set/) and [collation](https://www.mysqltutorial.org/mysql-collation/) depend on the settings of the client’s connection. The following table illustrates the specifiers and their meanings that you can use to construct a date format string:

| **Specifier** | **Meaning** |
| --- | --- |
| %a | Three-characters abbreviated weekday name e.g., Mon, Tue, Wed, etc. |
| %b | Three-characters abbreviated month name e.g., Jan, Feb, Mar, etc. |
| %c | Month in numeric e.g., 1, 2, 3…12 |
| %D | Day of the month with English suffix e.g., 0th, 1st, 2nd, etc. |
| %d | Day of the month with leading zero if it is 1 number e.g., 00, 01,02, …31 |
| %e | Day of the month without leading zero e.g., 1, 2,…31 |
| %f | Microseconds in the range of 000000.999999 |
| %H | Hour in 24-hour format with leading zero e.g., 00..23 |
| %h | Hour in 12-hour format with leading zero e.g., 01, 02…12 |
| %I | Same as %h |
| %i | Minutes with leading zero e.g., 00, 01,…59 |
| %j | Day of year with leading zero e.g., 001,002,…366 |
| %k | Hour in 24-hour format without leading zero e.g., 0,1,2…23 |
| %l | Hour in 12-hour format without leading zero e.g., 1,2…12 |
| %M | Full month name e.g., January, February,…December |
| %m | Month name with leading zero e.g., 00,01,02,…12 |
| %p | AM or PM, depending on other time specifiers |
| %r | Time in 12-hour format hh:mm:ss AM or PM |
| %S | Seconds with leading zero 00,01,…59 |
| %s | Same as %S |
| %T | Time in 24-hour format hh:mm:ss |
| %U | Week number with leading zero when the first day of week is Sunday e.g., 00,01,02…53 |
| %u | Week number with leading zero when the first day of week is Monday e.g., 00,01,02…53 |
| %V | Same as %U; it is used with %X |
| %v | Same as %u; it is used with %x |
| %W | Full name of weekday e.g., Sunday, Monday,…, Saturday |
| %w | Weekday in number (0=Sunday, 1= Monday,etc.) |
| %X | Year for the week in four digits where the first day of the week is Sunday; often used with %V |
| %x | Year for the week, where the first day of the week is Monday, four digits; used with %v |
| %Y | Four digits year e.g., 2000 and 2001. |
| %y | Two digits year e.g., 10, 11, and 12. |
| %% | Add percentage (%) character to the output |

The following are some commonly used date format strings:

| **DATE\_FORMAT string** | **Formatted date** |
| --- | --- |
| %Y-%m-%d | 2013-07-04 |
| %e/%c/%Y | 4/7/2013 |
| %c/%e/%Y | 7/4/2013 |
| %d/%m/%Y | 4/7/2013 |
| %m/%d/%Y | 7/4/2013 |
| %e/%c/%Y %H:%i | 4/7/2013 11:20 |
| %c/%e/%Y %H:%i | 7/4/2013 11:20 |
| %d/%m/%Y %H:%i | 4/7/2013 11:20 |
| %m/%d/%Y %H:%i | 7/4/2013 11:20 |
| %e/%c/%Y %T | 4/7/2013 11:20 |
| %c/%e/%Y %T | 7/4/2013 11:20 |
| %d/%m/%Y %T | 4/7/2013 11:20 |
| %m/%d/%Y %T | 7/4/2013 11:20 |
| %a %D %b %Y | Thu 4th Jul 2013 |
| %a %D %b %Y %H:%i | Thu 4th Jul 2013 11:20 |
| %a %D %b %Y %T | Thu 4th Jul 2013 11:20:05 |
| %a %b %e %Y | Thu Jul 4 2013 |
| %a %b %e %Y %H:%i | Thu Jul 4 2013 11:20 |
| %a %b %e %Y %T | Thu Jul 4 2013 11:20:05 |
| %W %D %M %Y | Thursday 4th July 2013 |
| %W %D %M %Y %H:%i | Thursday 4th July 2013 11:20 |
| %W %D %M %Y %T | Thursday 4th July 2013 11:20:05 |
| %l:%i %p %b %e, %Y | 7/4/2013 11:20 |
| %M %e, %Y | 4-Jul-13 |
| %a, %d %b %Y %T | Thu, 04 Jul 2013 11:20:05 |

* **Examples:**
* SELECT orderNumber, DATE\_FORMAT(orderdate, '%Y-%m-%d') orderDate, DATE\_FORMAT(requireddate, '%a %D %b %Y') requireddate, DATE\_FORMAT(shippedDate, '%W %D %M %Y') shippedDate FROM orders;
* MySQL DATE\_FORMAT with ORDER BY : SELECT orderNumber, DATE\_FORMAT(shippeddate, '%W %D %M %Y') shippeddate

FROM orders WHERE shippeddate IS NOT NULL ORDER BY shippeddate;

### DAYNAME Function

* MySQL DAYNAME function returns the name of a weekday for a specified [date](https://www.mysqltutorial.org/mysql-date/). The following illustrates the syntax of the DAYNAME function:

DAYNAME (date);

* The DAYNAME function accepts 1 argument which is a date that you want to get the name of its weekday.
* If the date is [NULL](https://www.mysqltutorial.org/mysql-null/) or invalid e.g., 2017-02-30, the DAYNAME function returns NULL.
* **Examples:**
* [SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) DAYNAME ('2000-01-01') dayname;
* SELECT DAYNAME (orderDate) weekday, COUNT (\*) total\_orders FROM orders WHERE YEAR(orderDate) = 2004 GROUP BY weekday ORDER BY total\_orders DESC;

### DAYOFWEEK Function

* The DAYOFWEEK function returns the weekday index for a [date](https://www.mysqltutorial.org/mysql-date/)i.e., 1 for Sunday, 2 for Monday, and 7 for Saturday.
* The following illustrates the DAYOFWEEK function: DAYOFWEEK(date)
* The DAYOFWEEK function accepts 1 argument which is a [DATE](https://www.mysqltutorial.org/mysql-date/) or [DATETIME](https://www.mysqltutorial.org/mysql-datetime/) value. It returns an integer which ranges from 1 to 7 that represents Sunday to Saturday. The DAYOFWEEK function returns [NULL](https://www.mysqltutorial.org/mysql-null/) if the date is NULL, zero (0000-00-00), or invalid.
* The following example returns weekday index of December 1st, 2010: [SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) DAYNAME('2012-12-01'), DAYOFWEEK('2012-12-01');

### EXTRACT Function

* The EXTRACT () function extracts part of a date. The following illustrates the syntax of the EXTRACT () function: EXTRACT (unit FROM date).
* The EXTRACT () function requires two arguments unit and date. The unit is the [interval](https://www.mysqltutorial.org/mysql-interval/) that you want to extract from the date. The following are the valid intervals for the unit argument : DAY, DAY\_HOUR, DAY\_MICROSECOND, DAY\_MINUTE, DAY\_SECOND, HOUR, HOUR\_MICROSECOND, HOUR\_MINUTE, HOUR\_SECOND,MICROSECOND, MINUTE, MINUTE\_MICROSECOND, MINUTE\_SECOND, MONTH, QUARTER, SECOND, SECOND\_MICROSECOND, WEE, YEAR, YEAR\_MONTH
* The date is a DATE or DATETIME value from which you extract an interval.
* **Examples:**
* Extract day from a date time: SELECT EXTRACT(DAY FROM '2017-07-14 09:04:44') DAY;
* Extract day\_hour from a date time: [SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) EXTRACT(DAY\_HOUR FROM '2017-07-14 09:04:44') DAYHOUR;
* Extract day\_microsecond from a date time: SELECT EXTRACT(DAY\_MICROSECOND FROM '2017-07-14 09:04:44') DAY\_MS;
* Extract hour from a datetime: SELECT EXTRACT(HOUR FROM '2017-07-14 09:04:44') HOUR;
* Extract month from a datetime: SELECT EXTRACT(MONTH FROM '2017-07-14 09:04:44') MONTH;
* Extract year from a datetime: SELECT EXTRACT(YEAR FROM '2017-07-14 09:04:44') YEAR;
* Extract year\_month from a date time: SELECT EXTRACT(YEAR\_MONTH FROM '2017-07-14 09:04:44') YEARMONTH;

### LAST\_DAY Function

* The LAST\_DAY () function takes a [DATE](https://www.mysqltutorial.org/mysql-date/) or [DATETIME](https://www.mysqltutorial.org/mysql-datetime/) value and returns the last day of the month for the input date.

# SQL Having Clause

* The HAVING Clause enables you to specify conditions that filter which group results appear in the results. The WHERE clause places conditions on the selected columns, whereas the HAVING clause places conditions on groups created by the GROUP BY clause.
* **Syntax**: The following code block shows the position of the HAVING Clause in a query.

SELECT FROM WHERE GROUP BY HAVING ORDER BY

* 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;

* Example:
* SELECT ID, NAME, AGE, ADDRESS, SALARY FROM CUSTOMERS GROUP BY age HAVING COUNT (age) >= 2;
* SELECT customer\_id, sum (Purchase\_amt) as "Total Amount", min (Purchase\_amt) as "Minimum Amount", MAX(Purchase\_amt) as "Maximum amount", AVG(Purchase\_amt) as "Average Amount", COUNT(Purchase\_amt) as "Number of orders" FROM `orders` GROUP by Customer\_ID having count (Customer\_ID)>=2 ORDER by Customer\_ID desc
* [SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) City, [COUNT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/group-by-functions.html#function_count)(Cust\_Id) as 'No of customers' from customer GROUP BY City HAVING [count](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/group-by-functions.html#function_count)(Cust\_Id)>1;

# SQL - Sub Queries

* A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause. A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.
* Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.
* There are a few rules that subqueries must follow –
* Subqueries must be enclosed within parentheses.
* A subquery can have only one column in the SELECT clause unless multiple columns are in the main query for the subquery to compare its selected columns.
* An ORDER BY command cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY command can be used to perform the same function as the ORDER BY in a subquery.
* The BETWEEN operator cannot be used with a subquery. However, the BETWEEN operator can be used within the subquery.

## Subqueries with the SELECT Statement

* Subqueries are most frequently used with the SELECT statement.
* 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:** [SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) \* FROM CUSTOMER WHERE customer\_id [IN](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/comparison-operators.html#function_in) ([SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) customer\_id FROM CUSTOMER WHERE credit >10000);

## 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\_copy table, you can use the following syntax.

INSERT INTO CUSTOMER\_copy SELECT \* FROM CUSTOMER WHERE customer\_id IN (SELECT customer\_id FROM CUSTOMER);

## 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 10 times in the CUSTOMERS table for all the customers whose city is Moscow.

UPDATE CUSTOMER SET credit = credit \* 10 WHERE city IN (SELECT city FROM CUSTOMER\_copy WHERE city ='Moscow' );

## 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.

DELETE FROM CUSTOMERS WHERE AGE IN (SELECT AGE FROM CUSTOMERS\_BKP WHERE AGE >= 27);

# SQL - USING Clause

* USING Clause is used to match only one column when more than one column matches. It should not have a qualifier (table name or Alias) in the referenced columns. NATURAL JOIN uses all the columns with matching names and datatypes to join the tables. The USING Clause can be used to specify only those columns that should be used for an EQUIJOIN.
* **Example 1:** Write SQL query to find the working location of the employees. Also give their respective employee\_id and last\_name?

[SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) e.EMP\_ID, e.emp\_name, d.dep\_location FROM Employees e JOIN Department d USING (dep\_id);

The example shown joins the DEPARTMENT\_ID column in the EMPLOYEES and DEPARTMENTS tables, and thus shows the location where an employee works.

* **Example 2:** Write SQL query to find the employee\_id, employee\_name, department\_name and department\_location.

[SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) e.emp\_id, e.emp\_name, d.dep\_name, d.dep\_location FROM employees e JOIN department d USING (dep\_id);

# SQL - 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. JOIN keeps the base tables (structure and data) unchanged.

## Join vs. Subquery.

* JOINs are faster than a subquery and it is very rare that the opposite.
* In JOINs the RDBMS calculates an execution plan, that can predict, what data should be loaded and how much it will take to processed and as a result this process save sometimes, unlike the subquery there is no pre-process calculation and run all the queries and load all their data to do the processing.
* A JOIN is checked conditions first and then put it into table and displays, where as a subquery take separate temp table internally and checking condition.
* When joins are using, there should be connection between two or more than two tables and each table has a relation with other while subquery means query inside another query, has no need to relation, it works on columns and conditions.

## INNER JOIN

* The [INNER JOIN](https://www.tutorialspoint.com/sql/sql-inner-joins.htm) - returns rows when there is a match in both tables. The most important and frequently used of the joins is the **INNER JOIN**. They are also referred to as an **EQUIJOIN**. The INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate. The query compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row.
* **Syntax**: SELECT table1.column1, table2.column2...FROM table1 INNER JOIN table2 ON table1.common\_field = table2.common\_field;
* **Example:**[SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) c.customer\_id as "Customer\_Customer\_id",o.customer\_id as "Orders\_customer\_id", c.customer\_name, o.purchase\_amt FROM CUSTOMER c INNER JOIN orders o on c.customer\_id = o.CUSTOMER\_ID;

## [LEFT JOIN](https://www.tutorialspoint.com/sql/sql-left-joins.htm)

* The SQL **LEFT JOIN** returns all rows from the left table, even if there are no matches in the right table. This means that if the ON clause matches 0 (zero) records in the right table; the join will still return a row in the result, but with NULL in each column from the right table. This means that a left join returns all the values from the left table, plus matched values from the right table or NULL in case of no matching join predicate.
* **Syntax:** The basic syntax of a LEFT JOIN is as follows.

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

* **Example:** SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS LEFT JOIN ORDERS ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

## [RIGHT JOIN](https://www.tutorialspoint.com/sql/sql-right-joins.htm)

* The SQL **RIGHT JOIN** returns all rows from the right table, even if there are no matches in the left table. This means that if the ON clause matches 0 (zero) records in the left table; the join will still return a row in the result, but with NULL in each column from the left table. This means that a right join returns all the values from the right table, plus matched values from the left table or NULL in case of no matching join predicate.
* **Syntax:** The basic syntax of a RIGHT JOIN is as follow.

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

* Example: MySQL RIGHT JOIN

In the following example *bval1* and *aval1* of both the associated table have matched and all the specified columns for matching from both the table have appeared. The unmatched row from the right table i.e., table111 have appeared and for those rows, the columns of left table i.e., table112 have set a value NULL. For each aval1 from the table table111 MySQL scans the left table - table112 to find the matching with bval1. When it finds the matching aval1 and bval1 it returns the other specified columns. For unmatched rows it returns null. Here, from the above example, it returns NULL for the value of aval1 200 and 400 because it does not exist in left table.

## SELF JOIN

* It 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.
* **Syntax:** The basic syntax of SELF JOIN is as follows

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

* Example:

SELECT a.ID, b.NAME, a. SALARY FROM CUSTOMERS a, CUSTOMERS b WHERE a. SALARY < b. SALARY;

# Stored Procedures

A stored procedure is a set of [Structured Query Language (SQL)](https://searchsqlserver.techtarget.com/definition/SQL) statements with an assigned name, which are stored in a [relational database management system](https://searchdatamanagement.techtarget.com/definition/RDBMS-relational-database-management-system) as a group, so it can be reused and shared by multiple programs. Stored procedures in SQL can accept input parameters and return multiple values of output parameters; in SQL Server, stored procedures program statements to perform operations in the database and return a status value to a calling procedure. If you want to save this query on the database server for execution later, one way to do it is to use a stored procedure. By definition, a stored procedure is a segment of declarative SQL statements stored inside the MySQL

## CREATE PROCEDURE

* To create a new stored procedure, you use the CREATE PROCEDURE statement. Here is the basic syntax of the CREATE PROCEDURE statement:

CREATE PROCEDURE procedure\_name(parameter\_list)

BEGIN

statements;

END //

* In this syntax

1. Specify the name of the stored procedure that you want to create after the CREATE PROCEDURE keywords.
2. Specify a list of comma-separated parameters for the stored procedure in parentheses after the procedure name.
3. Write the code between the BEGIN END block. The above example just has a simple [SELECT](https://www.mysqltutorial.org/mysql-select-statement-query-data.aspx) statement.

After the END keyword, you place the delimiter character to end the procedure statement.

## DROP PROCEDURE Statement

* The DROP PROCEDURE deletes a stored procedure from the database. The following shows the syntax of the DROP PROCEDURE statement:

DROP PROCEDURE [IF EXISTS] stored\_procedure\_name;

* In this syntax:

1. Specify the name of the stored procedure that you want to remove after the DROP PROCEDURE keywords.
2. Use IF EXISTS option to conditionally drop the stored procedure if it exists only.

## Stored Procedure Parameters

Almost stored procedures that you develop require parameters. The parameters make the stored procedure more flexible and useful. In MySQL, a parameter has one of three modes: IN, OUT, or INOUT.

### IN parameters

IN is the default mode. When you define an IN parameter in a stored procedure, the calling program must pass an argument to the stored procedure. In addition, the value of an IN parameter is protected. It means that even the value of the IN parameter is changed inside the stored procedure, its original value is retained after the stored procedure ends. In other words, the stored procedure only works on the copy of the IN parameter.

* **The IN - Parameter Example:** The following example creates a stored procedure that finds the city of the customer specified by the input parameter cityName:

DELIMITER //

CREATE PROCEDURE GetCustomerDetailsByCity (IN cityName VARCHAR (255))

BEGIN

SELECT \* FROM customer WHERE City = cityName;

END //

DELIMITER;

### OUT parameters

The value of an OUT parameter can be changed inside the stored procedure and its new value is passed back to the calling program. Notice that the stored procedure cannot access the initial value of the OUT parameter when it starts.

* **The OUT - Parameter Example:** The following stored procedure returns the number of orders by order status.

DELIMITER $$

CREATE PROCEDURE GetOrderCountBySalesmanId (IN sid int, OUT totalOrders INT)

BEGIN

SELECT COUNT (order\_no) INTO totalOrders FROM orders WHERE salesman\_id = sid;

END $$

DELIMITER;

* The stored procedure GetOrderCountBySalesmanId () has two parameters:

sid: is the IN parameter specifies the status of orders to return.

totalOrders: is the OUT parameter that stores the number of orders in a specific salesmanid.

To find the number of orders that already shipped, you call GetOrderCountBySalesmanId (), and also pass a session variable (@totalOrders) to receive the return value.

### INOUT parameters

An INOUT parameter is a combination of IN and OUT parameters. It means that the calling program may pass the argument, and the stored procedure can modify the INOUT parameter, and pass the new value back to the calling program.

## IF Statement

The IF statement has three forms: simple IF-THEN statement, IF-THEN-ELSE statement, and IF-THEN-ELSEIF- ELSE statement.

### Simple IF-THEN Statement

* The IF-THEN statement allows you to execute a set of SQL statements based on a specified condition.
* The following illustrates the **syntax** of the IF-THEN statement:

IF condition THEN

Statements;

END IF;

* In this syntax:

First, specify a condition to execute the code between the IF-THEN and END IF. If the condition evaluates to TRUE, the statements between IF-THEN and END IF will execute. Otherwise, the control is passed to the next statement following the END IF. Second, specify the code that will execute if the condition evaluates to TRUE.

* **Example to see the following GetCustomerLevel () stored procedure.**

DELIMITER $$

CREATE PROCEDURE GetCustomerLevel (IN pCustomerId INT, OUT pCustomerLevel VARCHAR (20))

BEGIN

DECLARE creditLimit DECIMAL (10, 2) DEFAULT 0;

SELECT credit INTO creditLimit FROM customer WHERE customer\_id = pCustomerId;

IF creditLimit > 5000 THEN

SET pCustomerLevel = 'PLATINUM';

END IF;

END $$

DELIMITER ;

* The stored procedure GetCustomerLevel () accepts two parameters: pCustomerId and pCustomerLevel.
* First, select credit of the customer specified by the pCustomerId from the customer table and store it in the local variable creditLimit.
* Then, set value for the OUT parameter pCustomerLevel to PLATINUM if the credit limit of the customer is greater than 5000.

### IF-THEN-ELSE statement

* In case you want to execute other statements when the condition in the IF branch does not evaluate to TRUE, you can use the IF-THEN-ELSE statement as follows. In this **syntax**, if the condition evaluates to TRUE, the statements between IF-THEN and ELSE execute. Otherwise, the else statements between the ELSE and END IF execute.

IF condition THEN

Statements;

ELSE

Else-statements;

END IF;

* **Example to see the following GetCustomerLevel () stored procedure.**

CREATE PROCEDURE GetCustomerLevel (IN pCustomerId INT, OUT pCustomerLevel VARCHAR (20))

BEGIN

DECLARE creditLimit DECIMAL (10, 2) DEFAULT 0;

SELECT credit INTO creditLimit FROM customer WHERE customer\_id = pCustomerId;

IF creditLimit > 5000 THEN

SET pCustomerLevel = 'PLATINUM';

ELSE

SET pCustomerLevel = 'NOT PLATINUM';

END IF;

END $$

DELIMITER ;

### IF-THEN-ELSEIF-ELSE statement

* If you want to execute statements conditionally based on multiple conditions, you use the following IF-THEN-ELSEIF-ELSE statement:
* **Syntax**

IF condition THEN

Statements;

ELSEIF elseif-condition THEN

Elseif-statements;

…

ELSE

Else-statements;

END IF;

* **Example to see the following GetCustomerLevel () stored procedure.**

DELIMITER $$

CREATE PROCEDURE GetCustomerLevel (IN pCustomerId INT, OUT pCustomerLevel VARCHAR (20))

BEGIN

DECLARE creditLimit DECIMAL (10, 2) DEFAULT 0;

SELECT credit INTO creditLimit FROM customer WHERE customer\_id = pCustomerId;

IF creditLimit > 5000 THEN

SET pCustomerLevel = 'PLATINUM';

ELSEIF credit <= 5000 AND credit > 1000 THEN

SET pCustomerLevel = 'GOLD';

ELSE

SET pCustomerLevel = 'SILVER';

END IF;

END $$

DELIMITER ;

## CASE Statement

Besides the [IF](https://www.mysqltutorial.org/mysql-if-statement/) statement, MySQL provides an alternative conditional statement called the CASE statement for constructing conditional statements in stored procedures. The CASE statements make the code more readable and efficient. The CASE statement has two forms: simpleCASE and searched CASE statements.

### Simple CASE statement

* The following is the basic **syntax** of the simple CASE statement:

CASE case\_value

WHEN when\_value1 THEN statements

WHEN when\_value2 THEN statements

...

[ELSE else-statements]

END CASE;

* In this syntax, the simple CASE statement sequentially compares the case\_value is with the when\_value1, when\_value2, until it finds one is equal. When the CASE finds a case\_value equal to a when\_value, it executes statements in the corresponding THEN clause. If CASE cannot find any when\_value equal to the case\_value, it executes the else statements in the ELSE clause if the ELSE clause is available. When the ELSE clause does not exist and the CASE cannot find any when\_value equal to the case\_value, it issues an error:
* **Simple CASE statement example:** The following stored procedure illustrates how to use the simple CASE statement

DELIMITER $$

CREATE PROCEDURE GetCustomerShipping (IN pCustomerNumber INT, OUT pShipping VARCHAR (50))

BEGIN

DECLARE customerCity VARCHAR (100);

SELECT city INTO customerCity FROM customer WHERE customer\_id = pCustomerNumber;

CASE customerCity

WHEN ‘California’ THEN SET pShipping = '2-day Shipping';

WHEN 'London' THEN SET pShipping = '3-day Shipping';

ELSE SET pShipping = '5-day Shipping';

END CASE;

END$$

DELIMITER;

* The GetCustomerShipping () stored procedure accepts two parameters: pCustomerNumber as an IN parameter and pShipping as an OUT parameter.
* In the stored procedure: First, select the city of the customer from the customers table by the input customer number. Second, use the simple CASE statement to determine the shipping time based on the country of the customer. If the customer locates in California, the shipping time is 2-day shipping. If the customer locates in London, the shipping time is 3-day shipping. The customers from other countries have 5-day shipping.

### Searched CASE statement

* The simple CASE statement only allows you to compare a value with a set of distinct values. To perform more complex matches such as ranges, you use the searched CASE statement. The searched CASE statement is equivalent to the IF statement, however, it’s much more readable than the IF statement.
* **Here is the basic syntax of the searched CASE statement:**

CASE

WHEN search\_condition1 THEN statements

WHEN search\_condition1 THEN statements

…

[ELSE else-statements]

END CASE;

* In this syntax, searched CASE evaluates each search\_condition in the WHEN clause until it finds a condition that evaluates to TRUE , then it executes the corresponding THEN clause statements. If no search\_condition evaluates to TRUE, the CASE will execute else-statements in the ELSE clause if an ELSE clause is available. Like the simple CASE statement, if you don’t specify an ELSE clause and no condition is TRUE, MySQL raises the same error: **Case** not found **for** **CASE** statement
* MySQL also does not allow you to have empty statements in the THEN or ELSE clause. If you don’t want to handle the logic in the ELSE clause while preventing MySQL from raising an error in case no search\_condition is true, you can use an empty BEGIN END block in the ELSE clause.
* **Searched CASE statement example1 :**

DELIMITER $$

CREATE PROCEDURE SearchCaseDemo (IN pCustomerId INT, OUT pGradeStatus VARCHAR (100))

BEGIN

DECLARE tempGrade INT DEFAULT 0;

SELECT grade into tempGrade from customer where customer\_id = pCustomerId;

CASE

WHEN tempGrade = 100 THEN

SET pGradeStatus = 'Grade 100';

WHEN tempGrade > 100 AND tempGrade <= 200 THEN

SET pGradeStatus = 'Grade lies between 100 and 200';

WHEN tempGrade >= 300 THEN

SET pGradeStatus = 'Grade is greater than equals 300';

ELSE

SET pGradeStatus = 'No Information';

END CASE;

END $$

DELIMITER ;

* **Searched CASE statement example2:**

DELIMITER $$

**CREATE** **PROCEDURE** GetDeliveryStatus(**IN** pOrderNumber INT, **OUT** pDeliveryStatus VARCHAR(100))

**BEGIN**

**DECLARE** waitingDay INT **DEFAULT** 0;

**SELECT** **DATEDIFF**(requiredDate, shippedDate)**INTO** waitingDay **FROM** orders **WHERE** orderNumber = pOrderNumber;

CASE

WHEN waitingDay = 0 THEN **SET** pDeliveryStatus = 'On Time';

WHEN waitingDay >= 1 AND waitingDay < 5 THEN **SET** pDeliveryStatus = 'Late';

WHEN waitingDay >= 5 THEN **SET** pDeliveryStatus = 'Very Late';

ELSE **SET** pDeliveryStatus = 'No Information';

**END** **CASE**;

**END**$$

DELIMITER ;

* The stored procedure GetDeliveryStatus() accepts an order number as an IN parameter and returns the delivery status as an OUT parameter. First, calculate the number of days between the required date and shipped date.
* Second, determine the delivery status based on the number of waiting days using the searched CASE statement:
  + If the number of waiting days is zero, then the delivery is on time.
  + When the number of waiting days is between 1 and 5, the delivery is late.
  + When the number of waiting days is more than 5 days, then the delivery is very late.
  + If the number of waiting days is NULL or else, the delivery has the status of no information specified in the ELSE clause.

## MySQL LOOP

* The LOOP statement allows you to execute one or more statements repeatedly.
* **Here is the basic syntax of the LOOP statement:**

[begin\_label:] LOOP

statement\_list

END LOOP [end\_label]

* The LOOP can have optional labels at the beginning and end of the block. The LOOP executes the statement\_list repeatedly. The statement\_list may have one or more statements, each terminated by a semicolon (;) statement delimiter. Typically, you terminate the loop when a condition is satisfied by using the [LEAVE](https://www.mysqltutorial.org/mysql-stored-procedure/mysql-leave/) statement. This is the typical syntax of the LOOP statement used with LEAVE statement:

[label]: LOOP

...

-- terminate the loop

IF condition THEN

LEAVE [label];

END IF;

...

END LOOP;

* The LEAVE statement immediately exits the loop. It works like the break statement in other programming languages like PHP, C/C++, and Java. In addition to the LEAVE statement, you can use the ITERATE statement to skip the current loop iteration and start a new iteration. The ITERATE is like the continue statement in PHP, C/C++, and Java.
* **MySQL LOOP statement example:** The following statement [creates a stored procedure](https://www.mysqltutorial.org/getting-started-with-mysql-stored-procedures.aspx) that uses a LOOP loop statement:

DELIMITER $$

CREATE PROCEDURE LoopExample()

BEGIN

DECLARE x int;

DECLARE strValue varchar(20);

SET x = 1;

SET strValue = '';

loopLabel : LOOP

IF x>10 THEN

LEAVE loopLabel;

END IF;

SET x = x + 1;

IF(x MOD 2) = 0 THEN

ITERATE loopLabel;

ELSE

SET strValue = concat(strValue,x,',');

END IF;

END LOOP;

SELECT str;

END $$;

DELIMITER ;

In this example:

* The stored procedure constructs a string from the even numbers e.g., 2, 4, and 6.
* The loop\_label *before* the LOOPstatement for using with the ITERATE and LEAVE statements.
* If the value of x is greater than 10, the loop is terminated because of the LEAVEstatement.
* If the value of the x is an odd number, the ITERATE ignores everything below it and starts a new loop iteration.
* If the value of the x is an even number, the block in the ELSEstatement will build the result string from even numbers.

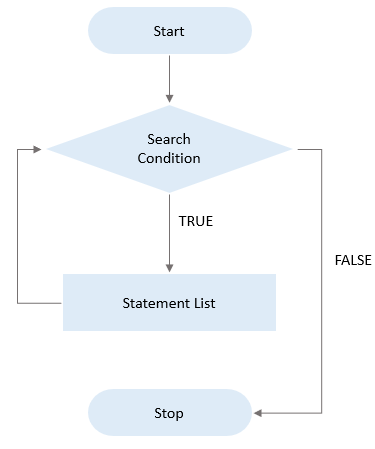
### MySQL WHILE Loop

* The WHILE loop is a loop statement that executes a block of code repeatedly as long as a condition is true. Here is the basic syntax of the WHILE statement:

[begin\_label:] WHILE search\_condition DO

statement\_list

END WHILE [end\_label]

* In this syntax:
* First, specify a search condition after the WHILE keyword.
* The WHILE checks the search\_condition at the beginning of each iteration.
* If the search\_condition evaluates to TRUE, the WHILE executes the statement\_list as long as the search\_condition is TRUE.
* The WHILE loop is called a pretest loop because it checks the search\_condition before the statement\_list executes.
* Second, specify one or more statements that will execute between the DO and END WHILE keywords.
* Third, specify optional labels for the WHILE statement at the beginning and end of the loop construct.
* The following flowchart illustrates the MySQL WHILE loop statement:
* **MySQL WHILE loop statement example:**
* First, [create a table](https://www.mysqltutorial.org/mysql-create-table/) named calendars which stores dates and derived date information such as day, month, quarter, and year:
* CREATE TABLE calendars (id INT AUTO\_INCREMENT, fulldate DATE UNIQUE, day TINYINT NOT NULL, month TINYINT NOT NULL, quarter TINYINT NOT NULL, year INT NOT NULL, PRIMARY KEY(id));
* Second, [create a new stored procedure](https://www.mysqltutorial.org/getting-started-with-mysql-stored-procedures.aspx) to [insert](https://www.mysqltutorial.org/mysql-insert-statement.aspx) a date into the calendars table:

DELIMITER $$

CREATE PROCEDURE InsertCalendar(dt DATE)

BEGIN

INSERT INTO calendars(fulldate,day,month,quarter,year) VALUES(dt, EXTRACT(DAY FROM dt),EXTRACT(MONTH FROM dt), EXTRACT(QUARTER FROM dt),EXTRACT(YEAR FROM dt));

END$$

DELIMITER ;

* Third, create a new stored procedure LoadCalendars() that loads a number of days starting from a start date into the calendars table.

DELIMITER $$

CREATE PROCEDURE LoadCalendars(startDate DATE, day INT)

BEGIN

DECLARE counter INT DEFAULT 1;

DECLARE dt DATE DEFAULT startDate;

WHILE counter <= day DO

CALL InsertCalendar(dt);

SET counter = counter + 1;

SET dt = DATE\_ADD(dt,INTERVAL 1 day);

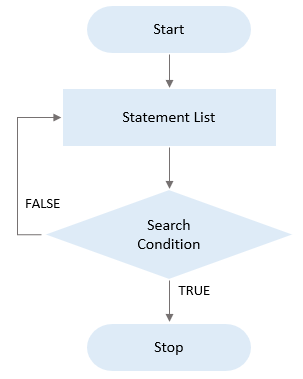
END WHILE;

END$$

DELIMITER ;

* The stored procedure LoadCalendars() accepts two arguments: startDate is the start date inserted into the calendars table.
* day is the number of days that will be loaded starting from the startDate.
* In the LoadCalendars() stored procedure: First, declare a counter and dt variables for keeping immediate values. The default values of counter and dt are 1 and startDate respectively. Then, check if the counter is less than or equal day, if yes: Call the stored procedure InsertCalendar() to insert a row into the calendars table. Increase the counter by one. Also, increase the dt by one day using the DATE\_ADD () function. The WHILE loop repeatedly inserts dates into the calendars table until the counter is equal to day.

## MySQL REPEAT Loop

* The REPEAT statement executes one or more statements until a search condition is true.
* Here is the basic syntax of the **REPEAT** loop statement:

[begin\_label:] REPEAT

statement

UNTIL search\_condition

END REPEAT [end\_label]

* The REPEAT executes the statement until the search\_condition evaluates to true.
* The REPEAT checks the search\_condition after the execution of statement, therefore, the statement always executes at least once. Therefore, the REPEAT is also known as a post-test loop.
* The REPEAT statement can have labels at the beginning and at the end. These labels are optional.
* **This statement**[**creates a stored procedure**](https://www.mysqltutorial.org/getting-started-with-mysql-stored-procedures.aspx)**called RepeatDemo that uses the REPEAT statement to concatenate numbers from 1 to 9:**

DELIMITER $$

CREATE PROCEDURE RepeatDemo()

BEGIN

DECLARE counter INT DEFAULT 1;

DECLARE result VARCHAR (100) DEFAULT '';

REPEAT

SET result = CONCAT (result,counter,',');

SET counter = counter + 1;

UNTIL counter >= 10

END REPEAT;

-- display result

SELECT result;

END$$

DELIMITER;

## MySQL LEAVE

* The LEAVE statement exits the flow control that has a given label. The following shows the basic syntax of the LEAVE statement:

LEAVE Label;

* In this syntax, you specify the label of the block that you want to exit after the LEAVE keyword.

### Using the LEAVE statement to exit a stored procedure

* If the label is the outermost of the [stored procedure](https://www.mysqltutorial.org/mysql-stored-procedure-tutorial.aspx)  or [function](https://www.mysqltutorial.org/mysql-stored-function/) block, LEAVE terminates the stored procedure or function. The following statement shows how to use the LEAVE statement to exit a stored procedure:

CREATE PROCEDURE sp\_name()

sp: BEGIN

IF condition THEN

LEAVE sp;

END IF;

-- other statement

END$$

* **EXAMPLE:**

DELIMITER $$

CREATE PROCEDURE demo(IN inCustomerNumber int)

sp: BEGIN

DECLARE customerCount INT;

DECLARE pStatus varchar(20);

-- check if the customer exists

SELECT COUNT(customer\_id) INTO customerCount FROM customer WHERE Customer\_Id = inCustomerNumber;

-- if the customer does not exist, terminate the stored procedure

IF customerCount = 0 THEN

SET pStatus = 'No such data available';

LEAVE sp;

ELSE

SELECT \* from customer;

SET pStatus = 'Data available';

END IF;

END $$

DELIMITER;

## Error Handling in Stored Procedures

* When an error occurs inside a stored procedure, it is important to handle it appropriately, such as continuing or exiting the current code block’s execution, and issuing a meaningful error message.
* MySQL provides an easy way to define handlers that handle from general conditions such as warnings or exceptions to specific conditions e.g., specific error codes.

### Declaring a handler

* To declare a handler, you use the DECLARE HANDLER statement as follows:

DECLARE action HANDLER FOR condition\_value statement;

* If a condition whose value matches the condition\_value, MySQL will execute the statement and continue or exit the current code block based on the action. The action accepts one of the following values:
* CONTINUE:  the execution of the enclosing code block (BEGIN … END) continues.
* EXIT: the execution of the enclosing code block, where the handler is declared, terminates.
* The condition\_value specifies a particular condition or a class of conditions that activate the handler. The  condition\_value accepts one of the following values:
* A MySQL error code.
* A standard SQLSTATE value. Or it can be an SQLWARNING, NOTFOUND or SQLEXCEPTION condition, which is shorthand for the class of SQLSTATE values. The NOTFOUND condition is used for a [cursor](https://www.mysqltutorial.org/mysql-cursor/)or SELECT INTO variable\_list statement.
* A named condition associated with either a MySQL error code or SQLSTATE value.
* The statement could be a simple statement or a compound statement enclosing by the BEGIN and END keywords.

# SQL Exercises, Practice, Solution

## Boolean, Relational, Wildcard and Special operators

1. Write a query to display all customers with a grade above 100.

* SELECT \* FROM customer WHERE grade > 100;

1. Write a query statement to display all customers in New York who have a grade value above 100.

* SELECT \* FROM customer WHERE city = 'New York' AND grade>100;

1. Write a SQL statement to display all customers, who are either belongs to the city New York or had a grade above 100.

* SELECT \* FROM customer WHERE city = 'New York' OR grade>100;

1. Write a SQL statement to display all the customers, who are either belongs to the city New York or not had a grade above 100.

* SELECT \* FROM customer WHERE city = 'New York' OR NOT grade>100;

1. Write a SQL query to display those customers who are neither belongs to the city New York nor grade value is more than 100.

* SELECT \* FROM customer WHERE NOT (city = 'New York' OR grade>100);

1. Write a SQL statement to display either those orders which are not issued on date 2012-09-10 and issued by the salesman whose ID is 5005 and below or those orders which purchase amount is 1000.00 and below.

* SELECT \* FROM orders WHERE NOT ((order\_date='2012-09-10' AND salesmanid<=5005) OR purchase\_amt<=1000.00);

1. Write a SQL statement to display salesman\_id, name, city and commission who gets the commission within the range more than 0.10% and less than 0.12%.

* SELECT salesman\_id, name, city, commission FROM salesman WHERE (commission > 0.10 AND commission< 0.12);

1. Write a SQL query to display all orders where purchase amount less than 200 or exclude those orders which order date is on or greater than 10th Feb, 2012 and customer id is below 3009.

* SELECT \* FROM orders WHERE (purch\_amt<200 OR NOT (ord\_date>='2012-02-10' AND customer\_id<3009));

1. Write a SQL statement to exclude the rows which satisfy 1) order dates are 2012-08-17 and purchase amount is below 1000 2) customer id is greater than 3005 and purchase amount is below 1000.

* SELECT \* FROM orders WHERE NOT ((ord\_date ='2012-08-17' OR customer\_id>3005) AND purch\_amt<1000);

1. Write a SQL statement to find those salesmen with all information who come from the city either Paris or Rome.

* SELECT \* FROM salesman WHERE city = 'Paris' OR city = 'Rome';

1. Write a query to filter those salesmen with all information who comes from any of the cities Paris and Rome.

* SELECT \* FROM salesman WHERE city IN ('Paris','Rome');

1. Write a query to produce a list of salesmen\_id, name, city and commission of each salesman who live in cities other than Paris and Rome.

* SELECT \* FROM salesman WHERE city NOT IN('Paris','Rome');

1. Write a query to sort out those customers with all information whose ID value is within any of 3007, 3008 and 3009.

* SELECT \* FROM customer WHERE customer\_id IN (3007, 3008, 3009);

1. Write a SQL statement to find those salesmen with all information who get the commission within a range of 0.12 and 0.14.

* SELECT \* FROM salesman WHERE commission BETWEEN 0.12 AND 0.14;

1. Write a query to filter all those orders with all information which purchase amount value is within the range 500 and 4000 except those orders of purchase amount value 948.50 and 1983.43.

* SELECT \* FROM orders WHERE (purch\_amt BETWEEN 500 AND 4000) AND NOT purch\_amt IN (948.50, 1983.43);

1. Write a SQL statement to find those salesmen with all other information and name started with any letter within 'A' and 'K'.

* SELECT \* FROM salesman WHERE name BETWEEN 'A' and 'L';

1. Write a SQL statement to find those salesmen with all other information and name started with other than any latter within 'A' and 'L'.

* SELECT \* FROM salesman WHERE name NOT BETWEEN 'A' and 'L';

1. Write a SQL statement to find that customers whose name begin with the letter 'B'.

* SELECT \* FROM customer WHERE cust\_name LIKE 'B%';

1. Write a SQL statement to find those salesmen with all information whose name containing the 1st character is 'N' and the 4th character is 'l' and rests may be any character.

* SELECT \* FROM salesman WHERE name LIKE 'N\_\_l%';

1. Write a query to display the orders according to the order number arranged by ascending order.

* SELECT \* FROM orders ORDER BY ord\_no;

## Aggregate Functions

1. Write a SQL statement to find the total purchase amount and average purchase amount of all orders.

* SELECT SUM (purch\_amt), AVG (purch\_amt) FROM orders;

1. Write a SQL statement to know how many customers have listed their names.

* SELECT COUNT (\*) FROM customer;

1. Write a SQL statement to find the number of salesmen currently listing for all of their customers.

* SELECT COUNT (DISTINCT salesman\_id) FROM orders;

1. Write a SQL statement to get the maximum purchase amount and minimum purchase amount of all the orders.

* SELECT MAX (purch\_amt), MIN(purch\_amt) FROM orders;

1. Write a SQL statement which selects the highest grade for each of the cities of the customers.

* SELECT city, MAX(grade) FROM customer GROUP BY city;

1. Write a SQL statement to find the highest purchase amount ordered by each customer with their ID and highest purchase amount.

* SELECT customer\_id, MAX(purch\_amt) FROM orders GROUP BY customer\_id;

1. Write a SQL statement to find the highest purchase amount ordered by each customer on a particular date with their ID, order date and highest purchase amount.

* SELECT customer\_id, ord\_date, MAX(purch\_amt) FROM orders GROUP BY customer\_id;

1. Write a SQL statement to find the highest purchase amount on a date '2012-08-17' for each salesman with their ID.

* SELECT salesman\_id, MAX(purch\_amt) FROM orders WHERE ord\_date = '2012-08-17' GROUP BY salesman\_id;

1. Write a SQL statement to find the highest purchase amount with their ID and order date, for only those customers who have highest purchase amount in a day is more than 2000.

* SELECT customer\_id, ord\_date, MAX(purch\_amt) FROM orders GROUP BY customer\_id HAVING MAX(purch\_amt)>2000.00;

1. Write a SQL statement to find the highest purchase amount with their ID and order date, for those customers who have a higher purchase amount in a day is within the range 2000 and 6000.

* SELECT customer\_id, ord\_date, MAX(purch\_amt) FROM orders GROUP BY customer\_id HAVING MAX(purch\_amt) BETWEEN 2000 AND 6000;

1. Write a SQL statement to find the highest purchase amount with their ID and order date, for only those customers who have a higher purchase amount in a day is within the list 2000, 3000, 5760 and 6000.

* SELECT customer\_id, ord\_date, MAX (purch\_amt) FROM orders GROUP BY customer\_id HAVING MAX(purch\_amt) IN(2000 ,3000,5760, 6000);

1. Write a SQL statement to find the highest purchase amount with their ID, for only those customers whose ID is within the range 3002 and 3007.

* SELECT customer\_id, MAX (purch\_amt) FROM orders WHERE customer\_id BETWEEN 3002 and 3007 order BY customer\_id desc;

1. Write a SQL statement to display customer details (ID and only highest purchase amount) who’s IDs are within the range 3002 and 3007 and highest purchase amount is more than 1000.

* SELECT customer\_id, MAX(purch\_amt) FROM orders WHERE customer\_id BETWEEN 3002 and 3007 GROUP BY customer\_id HAVING MAX(purch\_amt)>1000;

1. Write a SQL statement to find the highest purchase amount with their ID, for only those salesmen whose ID is within the range 5003 and 5008.

* SELECT salesman\_id, MAX(purch\_amt) FROM orders where salesman\_id BETWEEN 5003 AND 5008 GROUP BY salesman\_id;

1. Write a SQL statement that counts all orders for a date August 17th, 2012.

* SELECT COUNT(\*) FROM orders where ord\_date='2012-08-17';

1. Write a query that counts the number of salesmen with their order date and ID registering orders for each day.

* SELECT ord\_date, salesman\_id, COUNT (\*) FROM orders GROUP BY ord\_date, salesman\_id;

## Subqueries

1. Write a query to display all the orders for the salesman who belongs to the city London.

* SELECT \* FROM orders WHERE salesman\_id = (SELECT salesman\_id FROM salesman WHERE city='London');

1. Write a query to find all the orders issued against the salesman who may works for customer whose id is 3007.

* SELECT \* FROM orders WHERE salesman\_id = (SELECT DISTINCT salesman\_id FROM orders WHERE customer\_id =3007);

1. Write a query to display all the orders which values are greater than the average order value for 10th October 2012.

* SELECT \* FROM orders WHERE purchase\_amt > (SELECT AVG (purchase\_amt) FROM orders WHERE order\_date ='2012-10-10');

1. Write a query to find all orders attributed to a salesman in New York.

* SELECT \* FROM orders WHERE salesman\_id IN (SELECT salesman\_id FROM salesman WHERE city ='New York');

1. Write a query to display the commission of all the salesmen servicing customers in Paris.

* SELECT commission FROM salesman WHERE salesman\_id IN (SELECT salesman\_id FROM customer WHERE city = 'Paris');

1. Write a query to display all the customers whose id is 2001 below the salesman ID of Mc Lyon.

* SELECT \* FROM customer WHERE customer\_id = (SELECT salesman\_id - 2001 FROM salesman WHERE name = 'Mc Lyon');

1. Write a query to count the customers with grades above New York's average.

* [SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) grade,[COUNT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/group-by-functions.html#function_count)(grade) FROM customer GROUP BY grade HAVING grade > ([SELECT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/select.html) [AVG](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/group-by-functions.html#function_avg) (grade) FROM customer WHERE city = 'New York');

1. Write a query to extract the data from the orders table for those salesmen who earned the maximum commission.

* SELECT order\_no, purchase\_amt, order\_date, salesman\_id FROM orders WHERE salesman\_id IN (SELECT salesman\_id FROM salesman WHERE commission = (SELECT MAX (commission) FROM salesman));

1. Write a query to find all orders with order amounts which are above-average amounts for their customers.

* Select \* from orders where purchase\_amt> (select AVG (purchase\_amt) from orders);

1. Write a query to extract all data from the customer table if and only if one or more of the customers in the customer table are located in London.

* SELECT customer\_id,customer\_name, city FROM customer WHERE EXISTS (SELECT \* FROM customer WHERE city='London');

1. Write a query to find the salesmen who have multiple customers.

* select \* from salesman where salesman\_id in (SELECT salesman\_id FROM customer GROUP BY salesman\_id HAVING COUNT (salesman\_id) >= 2)

1. Write a query to find all the salesmen for whom there are customers that follow them.

* SELECT \* FROM salesman WHERE city IN (SELECT city FROM customer);

1. Write a query to display all orders with an amount smaller than the maximum amount for a customer in London.

* SELECT \* FROM orders WHERE purchase\_amt < (SELECT MAX (purchase\_amt) FROM orders a, customer b WHERE a.customer\_id=b.customer\_id AND b.city='London');

1. Write a query to find all those customers whose grade are not as the grade who belongs to the city Paris.

* SELECT \* FROM customer WHERE grade NOT IN (SELECT grade FROM customer WHERE city='Paris');

## JOINS

1. Write a SQL statement to prepare a list with salesman name, customer name and their cities for the salesmen and customer who belongs to the same city.

* SELECT salesman.name AS "Salesman", customer.cust\_name, customer.city FROM salesman inner join customer WHERE salesman.city = customer.city;

1. Write a SQL statement to make a list with order no, purchase amount, customer name and their cities for those orders which order amount between 500 and 2000.

* SELECT a.ord\_no,a.purch\_amt, b.cust\_name,b.city FROM orders a,customer b WHERE a.customer\_id = b.customer\_id AND a.purch\_amt BETWEEN 500 AND 2000;

1. Write a SQL statement to know which salesman are working for which customer.

* SELECT a. cust\_name AS "Customer Name", a. city, b.name AS "Salesman", b. commission FROM customer a INNER JOIN salesman b ON a.salesman\_id = b.salesman\_id;

1. Write a SQL statement to find the list of customers who appointed a salesman for their jobs who gets a commission from the company is more than 12%.

* SELECT a. cust\_name AS "Customer Name", a. city, b.name AS "Salesman", b. commission FROM customer a INNER JOIN salesman b ON a. salesman\_id = b. salesman\_id WHERE b. commission>.12;

1. Write a SQL statement to find the list of customers who appointed a salesman for their jobs who does not live in the same city where their customer lives, and gets a commission is above 12%.

* SELECT a.cust\_name AS "Customer Name", a.city, b.name AS "Salesman", b.city,b.commission FROM customer a INNER JOIN salesmanb ON a.salesman\_id = b.salesman\_id WHERE b.commission>.12 AND a.city<>b.city;

1. Write a SQL statement to find the details of a order i.e. order number, order date, amount of order, which customer gives the order and which salesman works for that customer and commission rate he gets for an order.

* SELECT a.ord\_no,a.ord\_date,a.purch\_amt,b.cust\_name AS "Customer Name", b.grade, c.name AS "Salesman", c.commission FROM orders a INNER JOIN customer b ON a.customer\_id=b.customer\_id INNER JOIN salesman c ON a.salesman\_id=c.salesman\_id;

1. Write a SQL statement to make a list in ascending order for the customer who works either through a salesman or by own.

* SELECT a.cust\_name,a.city,a.grade,b.name AS "Salesman",b.city FROM customer a LEFT JOIN salesman b ON a.salesman\_id =b.salesman\_id order by a.customer\_id;

1. Write a SQL statement to make a list in ascending order for the customer who holds a grade less than 300 and works either through a salesman or by own.

* SELECT a.cust\_name, a.city,a.grade, b.name AS "Salesman", b.city FROM customer a LEFT OUTER JOIN salesman b ON a.salesman\_id = b.salesman\_id WHERE a.grade<300 ORDER BY a.customer\_id;