**SQL COALESCE Function**

The MySQL COALESCE function returns the first non-NULL value from the given list. If there are no non-null values The COALESCE function returns NULL.

**Syntax:**

Following is the syntax to use the MySQL COALESCE function.



**Parameters:**  
**Expression1 to Expression\_n:** The expressions to test for non-null values.

**Return Value:** The COALESCE function returns the first non-null value in a list of expressions or NULL if all the values in a list are NULL.

**Examples:**

Let’s understand the COALESCE function with some examples and also understand how to use the COALESCE function in MySQL.

**Example1:**  
**SELECT COALESCE(null, null, ‘Apple’, ‘Ball’);**  
**Output: Apple**

**Example2:**  
**SELECT COALESCE(‘Apple’, ‘Ball’, null, ‘Cat’, ‘Dog’);**  
**Output: Apple**

**Example3:**  
**SELECT COALESCE(null, 10, 20, 30, null, 40);**  
**Output: 10**

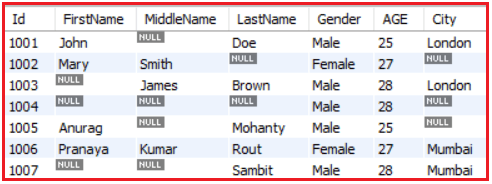
**Example4:**  
**SELECT COALESCE(null, ‘MySQLTutorials’, ‘dotnettutorials.net’);**  
**Output: MySQLTutorials**

**Example5:**  
**SELECT COALESCE(null, null, null, null, null);**  
**Output: NULL**

**Example6:**  
**SELECT COALESCE(‘dotnettutorials.net’, ‘Tutorials’, ‘MySQLTutorials’);**  
**Output: ‘dotnettutorials.net**

**COALESCE Function with Database Table:**

We are going to use the following Employee table to understand how to use the COALESCE function with the database table column with Examples.



Please use the following SQL Script to create and populate the Employee table with the required sample data.

**CREATE** **DATABASE** Company;

**USE** Company;

**CREATE** **TABLE** Employee (

Id **INT** **PRIMARY KEY**,

FirstName **VARCHAR**(45),

MiddleName **VARCHAR**(45),

LastName **VARCHAR**(45),

Gender **VARCHAR**(45) **NOT NULL**,

**AGE** **INT** **NOT NULL**,

City **VARCHAR**(45)

);

**INSERT** **INTO** Employee **VALUES** (1001, 'John', **NULL**, 'Doe', 'Male', 25, 'London');

**INSERT** **INTO** Employee **VALUES** (1002, 'Mary', 'Smith', **NULL**, 'Female', 27, **NULL**);

**INSERT** **INTO** Employee **VALUES** (1003, **NULL**, 'James', 'Brown', 'Male', 28, 'London');

**INSERT** **INTO** Employee **VALUES** (1004, **NULL**, **NULL**, **NULL**, 'Male', 28, **NULL**);

**INSERT** **INTO** Employee **VALUES** (1005, 'Anurag', **NULL**, 'Mohanty', 'Male', 25, **NULL**);

**INSERT** **INTO** Employee **VALUES** (1006, 'Pranaya', 'Kumar', 'Rout', 'Female', 27, 'Mumbai');

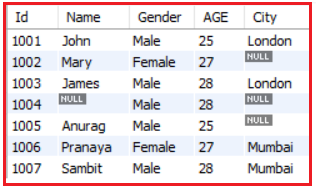
**INSERT** **INTO** Employee **VALUES** (1007, **NULL**, **NULL**, 'Sambit', 'Male', 28, 'Mumbai');

**Example1:**

In the below example, we use COALESCE function with First, Middle, and Last Name column values.

**SELECT Id, COALESCE(FirstName, MiddleName, LastName) AS Name, Gender, AGE, City FROM Employee;**

Once you execute the above query, you will get the following output.



**Example2:**

In the below example, we use COALESCE function with the City column. Instead of showing NULL, we are showing NA where the city column value is null.

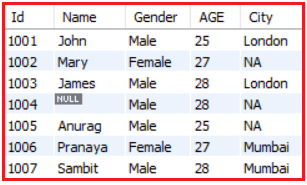
**SELECT** Id,

**COALESCE**(FirstName, MiddleName, LastName) **AS** Name,

Gender, **AGE**, **COALESCE**(City, 'NA') **AS** City

**FROM** Employee;

When you execute the above SQL Statement, you will get the following output.



**Note:** The COALESCE function accepts one parameter which is the list that can contain various values. The value returned by the COALESCE function is the first non-null value or NULL if all the values in a list are NULL.

**Correlated vs Non-Correlated Subquery**

Imagine you’re a detective trying to solve a case (the outer query). You have some clues (the data in the main table), but need extra information (the subquery) to crack it. Let’s talk about two ways you can gather that information.

**Non-correlated subquery:** This detective is independent. They gather their extra info (the subquery) once, like interviewing a witness, and use it for the whole case. Think of it like a reference book you consult throughout your investigation. It doesn’t depend on any specific clue you’re currently examining.

SELECT employee\_id, name, salary  
FROM employees  
WHERE salary > (SELECT AVG(salary) FROM employees);

You want to find all employees who earn more than the average salary. You would use a non-correlated subquery to first calculate the average salary (the extra info) and then compare each employee’s salary to it. The subquery only runs once, regardless of how many employees there are.

**Correlated subquery:** This detective is nosy and asks different questions for each clue (the outer query). They need specific details based on what they’re currently examining. Think of it like an interrogator who changes their questions depending on the suspect’s responses.

You want to find managers who have more employees under them than the average number of employees per manager. You would use a correlated subquery to count the number of employees for each manager (the extra info), compare it to the average, and only return those who exceed it. This subquery runs once for each manager you’re analyzing.

SELECT manager\_id, name  
FROM employees e  
WHERE   
(SELECT COUNT(\*) FROM employees WHERE manager\_id = e.employee\_id) >  
 (SELECT AVG(emp\_count)   
 FROM (SELECT manager\_id, COUNT(\*) AS emp\_count   
 FROM employees GROUP BY manager\_id) AS manager\_counts);

**Which One is More Faster ?**

Non-correlated subqueries generally tend to be faster than correlated subqueries due to the following reasons:

* Execution frequency: Non-correlated subqueries execute only once, regardless of the number of rows in the outer query. This reduces the overall processing time.

Correlated Subquery:

* **Dependency:** The inner subquery relies on values from the outer query's result.
* **Execution:** It's evaluated repeatedly, once for each row processed by the outer query.
* **Example:** A query that finds employees whose salary is greater than the average salary of their department, where the average is calculated for each department.
* **Performance:** Can be less efficient than uncorrelated subqueries due to repeated execution.

Uncorrelated Subquery:

* **Independence:**

The inner subquery is independent of the outer query and its results.

* **Execution:**

It's executed only once, providing a single result that's used by the outer query.

* **Example:**

A query that retrieves employees whose salary is greater than the overall average salary of all employees.

* **Performance:**

Generally more efficient than correlated subqueries because the inner query is executed only once.

Date & Time Function:

Handling **date and time** data in MySQL is essential for many database operations, especially when it comes to handling timestamps, scheduling tasks, or generating time-based. MySQL provides a variety of date and time functions that help users work with date values, perform calculations, and format them as needed.

These functions allow developers to perform calculations, extract specific parts of a date, or even format the output for better readability. In this article, we will explore the most commonly used MySQL date functions, explain their syntax, provide examples, and walk you through how to effectively use them

**Understanding MySQL Date and Time Data Types**

When working with dates in MySQL, it can be tricky for beginners, especially because the format of the date in the database must match the format of the input data during insertion. In many cases, instead of just using a simple date, we may need to store both a date and time, depending on the use case. This is where the **DATETIME**and **TIMESTAMP**data types come into play.

MySQL provides the following data types for storing date and time values:

**1. DATE:**

* **Format**: YYYY-MM-DD
* Used for storing only the date (year, month, and day), without any time component.

**2. DATETIME:**

* **Format**: YYYY-MM-DD HH:MM:SS
* Stores both the date and the time, making it useful when you need to capture specific times along with the date.

**3. TIMESTAMP:**

* **Format**: YYYY-MM-DD HH:MM:SS
* Similar to DATETIME, but TIMESTAMP also includes time zone support. It automatically updates to the current timestamp when a record is modified, making it ideal for tracking changes to records over time.

**4. YEAR:**

* **Format**: YYYY or YY
* Used for storing just the year, which can be useful for applications where only the year is needed, such as tracking birth years or fiscal years.

**MySQL Date and Time Functions**

Now, let's dive into the MySQL date functions that you can use to manipulate and query date and time data effectively.

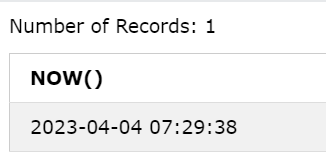
**1. NOW()- Get Current Date and Time**

The **NOW()** function retrieves the current date and time in **YYYY-MM-DD HH:MI:SS** format.

**Query:**

SELECT NOW();

**Output:**



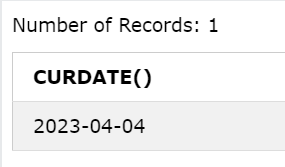
**2. CURDATE() - Get Current Date Only**

If we only need the current date without the time portion, we can use **CURDATE()**, which returns the date in **YYYY-MM-DD** format.

**Query:**

SELECT CURDATE();

**Output:**



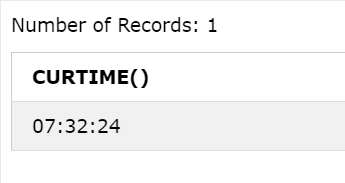
**3. CURTIME() - Get Current Time Only**

The CURTIME() function returns the current time in **HH:MI:SS** format, excluding the date.

**Query:**

SELECT CURTIME();

**Output:**



**4. DATE() - Extract the Date Part from a DATETIME Value**

The DATE() function extracts only the date part from a DATETIME or TIMESTAMP value, discarding the time.

**Example:** For a table called **users**:

| **Id** | **Name** | **BirthTime** |
| --- | --- | --- |
| 4120 | Pratik | 1996-09-26 16:44:15.581 |

**Query:**

SELECT Name, DATE(BirthTime)   
AS BirthDate FROM Test;

**Output:**

| **Name** | **BirthDate** |
| --- | --- |
| Pratik | 1996-09-26 |

**5. EXTRACT() - Extract Specific Date Parts**

EXTRACT() is used to extract specific parts of a date, such as the day, month, or year. This function can be especially helpful when analyzing or comparing different parts of a date. Several units can be considered but only some are used such as **MICROSECOND, SECOND, MINUTE, HOUR, DAY, WEEK, MONTH, QUARTER, YEAR, etc.** And 'date' is a valid date expression.

**Syntax:**

*EXTRACT(unit FROM date);*

**Query to Extract Day:**

SELECT Name, Extract(DAY FROM   
BirthTime) AS BirthDay FROM Test;

**Output:**

| **Name** | **Birthday** |
| --- | --- |
| Pratik | 26 |

**Query to Extract Year:**

SELECT Name, Extract(YEAR FROM BirthTime)  
AS BirthYear FROM Test;

**Output:**

| **Name** | **BirthYear** |
| --- | --- |
| Pratik | 1996 |

**Query to Extract Seconds:**

SELECT Name, Extract(SECOND FROM   
BirthTime) AS BirthSecond FROM Test;

**Output:**

| **Name** | **BirthSecond** |
| --- | --- |
| Pratik | 581 |

**6. DATE\_ADD() - Add Intervals to a Date**

The **DATE\_ADD()** function allows you to add time intervals (e.g., days, months, years) to a date or DATETIME value.

**Syntax:**

*DATE\_ADD(date, INTERVAL expr type);*

**Example:** For the below table named '**Test**'

| **Id** | **Name** | **BirthTime** |
| --- | --- | --- |
| 4120 | Pratik | 1996-09-26 16:44:15.581 |

**Query to Add 1 Year to a Date**

SELECT Name, DATE\_ADD(BirthTime, INTERVAL   
1 YEAR) AS BirthTimeModified FROM Test;

**Output:**

| **Name** | **BirthTimeModified** |
| --- | --- |
| Pratik | 1997-09-26 16:44:15.581 |

**Query to Add 30 days to a Date**

SELECT Name, DATE\_ADD(BirthTime,   
INTERVAL 30 DAY) AS BirthDayModified FROM Test;

**Output:**

| **Name** | **BirthDayModified** |
| --- | --- |
| Pratik | 1996-10-26 16:44:15.581 |

**Query to Add 4 Hours to a Date**

SELECT Name, DATE\_ADD(BirthTime, INTERVAL  
 4 HOUR) AS BirthHourModified FROM Test;

**Output:**

| **Name** | **BirthSecond** |
| --- | --- |
| Pratik | 1996-10-26 20:44:15.581 |

**7. DATEDIFF() – Find the Difference Between Two Dates**

This Function returns the number of days between two dates.

**Syntax:**

*DATEDIFF(interval,date1, date2);*

*interval - minute/hour/month/year,etc*

*date1 & date2- date/time expression*

**Query to Find the Difference Between Two Dates**

SELECT DATEDIFF(day, '2017-01-13', '2017-01-03') AS DateDiff;

**Output:**

| **DateDiff** |
| --- |
| 0 |

**8. DATE\_FORMAT() - Foramt Date and Time**

DATE\_FORMAT() allows us to format a **DATE**, **DATETIME**, or **TIMESTAMP**value into a custom format using placeholders.

**Syntax:**

*DATE\_FORMAT(date,format);*

the date is a valid date and the format specifies the output format for the date/time. The formats that can be used are:

* %a-Abbreviated weekday name (Sun-Sat)
* %b-Abbreviated month name (Jan-Dec)
* %c-Month, numeric (0-12)
* %D-Day of month with English suffix (0th, 1st, 2nd, 3rd)
* %d-Day of the month, numeric (00-31)
* %e-Day of the month, numeric (0-31)
* %f-Microseconds (000000-999999)
* %H-Hour (00-23)
* %h-Hour (01-12)
* %I-Hour (01-12)
* %i-Minutes, numeric (00-59)
* %j-Day of the year (001-366)
* %k-Hour (0-23)
* %l-Hour (1-12)
* %M-Month name (January-December)
* %m-Month, numeric (00-12)
* %p-AM or PM
* %r-Time, 12-hour (hh:mm: ss followed by AM or PM)
* %S-Seconds (00-59)
* %s-Seconds (00-59)
* %T-Time, 24-hour (hh:mm: ss)
* %U-Week (00-53) where Sunday is the first day of the week
* %u-Week (00-53) where Monday is the first day of the week
* %V-Week (01-53) where Sunday is the first day of the week, used with %X
* %v-Week (01-53) where Monday is the first day of the week, used with %x
* %W-Weekday name (Sunday-Saturday)
* %w-Day of the week (0=Sunday, 6=Saturday)
* %X-Year for the week where Sunday is the first day of the week, four digits, used with %V
* %x-Year for the week where Monday is the first day of the week, four digits, used with %v
* %Y-Year, numeric, four digits
* %y-Year, numeric, two digits

**Query to Format a Date**

SELECT DATE\_FORMAT('2025-04-10 12:34:56', '%W, %M %d, %Y') AS formatted\_date;

**Output:**

| **Formatted\_date** |
| --- |
| Thursday, April 10, 2025 |

**Best Practices for Working with Date and Time in MySQL**

**1. Always Use Proper Date Formats**

When inserting or updating dates, ensure that you follow MySQL’s expected formats (YYYY-MM-DD, YYYY-MM-DD HH:MI:SS). This ensures that your queries return expected results without errors.

**2. Time Zone Considerations with TIMESTAMP**

While DATETIME does not store timezone information, TIMESTAMP does. If your application is time zone sensitive (for example, in international applications), consider using TIMESTAMP for date-time fields that need to account for different time zones.

**3. Handling Date Ranges in Queries**

For filtering data based on date ranges (e.g., retrieving all records from a specific month or year), ensure you use proper date comparisons in the WHERE clause. Example:

SELECT \* FROM orders  
WHERE order\_date BETWEEN '2025-01-01' AND '2025-01-31';

**4. Performance Considerations**

When working with date and time functions, especially in large datasets, be mindful of the impact on performance. Avoid using functions like NOW() or CURDATE() in the WHERE clause, as they can slow down queries when dealing with large tables.

**🔁 What Is Function Nesting in SQL?**

**Function nesting** means calling one function **inside another**.

**📌 Example Syntax:**

OUTER\_FUNCTION(INNER\_FUNCTION(column))

**🎯 Real-Time Example: Customer Invoice System**

Suppose you have a table of **customer invoices**:

**Invoices Table:**

| **InvoiceID** | **CustomerName** | **Amount** | **Discount** | **TaxRate** | **DueDate** |
| --- | --- | --- | --- | --- | --- |
| 1 | Alice | 1000.00 | NULL | 0.10 | 2024-12-01 |
| 2 | Bob | 2000.00 | 150.00 | 0.08 | 2024-11-01 |
| 3 | Carol | 500.00 | NULL | 0.12 | 2024-10-15 |

**🧾 Problem:**

We want to:

* Treat NULL discounts as 0
* Apply tax on the **net amount after discount**
* Show how many days are left until the due date

**✅ Nested Functions Example Query:**

SELECT

InvoiceID,

CustomerName,

Amount,

COALESCE(Discount, 0) AS DiscountUsed,

ROUND((Amount - COALESCE(Discount, 0)) \* (1 + TaxRate), 2) AS FinalAmountWithTax,

DATEDIFF(DueDate, CURDATE()) AS DaysUntilDue

FROM

Invoices;

**🔍 How Functions Are Nested:**

| **Expression** | **Nesting Explanation** |
| --- | --- |
| COALESCE(Discount, 0) | Replaces NULL discount with 0 |
| Amount - COALESCE(...) | Subtracts discount from amount |
| (...) \* (1 + TaxRate) | Applies tax to net amount |
| ROUND(..., 2) | Rounds to 2 decimal places |
| DATEDIFF(DueDate, CURDATE()) | Calculates days until invoice is due |

**🖥️ Sample Output:**

| **InvoiceID** | **CustomerName** | **DiscountUsed** | **FinalAmountWithTax** | **DaysUntilDue** |
| --- | --- | --- | --- | --- |
| 1 | Alice | 0.00 | 1100.00 | 186 |
| 2 | Bob | 150.00 | 1998.00 | 155 |
| 3 | Carol | 0.00 | 560.00 | 138 |

**🧠 Other Useful Nesting Examples**

**🔸 Nesting String Functions**

SELECT UPPER(CONCAT(FirstName, ' ', LastName)) AS FullName

FROM Customers;

* Combines first and last names, then capitalizes the result.

**🔸 Nesting in Conditional Logic**

SELECT

CustomerID,

IFNULL(ROUND(TotalSpent / OrdersCount, 2), 0) AS AverageOrderValue

FROM CustomerStats;

* Divides total spent by order count, handles division by zero or NULLs.

**✅ Summary**

| **Feature** | **Example** |
| --- | --- |
| Nested Arithmetic | ROUND((Amount - COALESCE(Discount, 0)) \* 1.1, 2) |
| Nested String Ops | UPPER(CONCAT(...)) |
| Nested Date Ops | DATEDIFF(DueDate, CURDATE()) |
| Real Use | Invoices, dashboards, reporting |