Mathematical Functions

Function	Example	Description
ROUND()	`SELECT ROUND(price, 2) FROM Products;`	*Rounds the price column to two decimal places, often used for formatting currency.*
ABS()	`SELECT ABS(balance) FROM Accounts;`	*Removes the negative sign from any negative values in balance, often used to handle financial data consistently.*
MOD()	`SELECT MOD(quantity, 2) FROM Orders;`	*Calculates the remainder when quantity is divided by 2, useful for identifying even or odd numbers.*
CEIL()	`SELECT CEIL(price) FROM Products;`	*Rounds up the price to the nearest integer, useful for price estimations or minimum integer constraints.*
FLOOR()	`SELECT FLOOR(price) FROM Products;`	*Rounds down the price to the nearest integer, useful for reducing values to whole units without exceeding the original.*
POWER()	`SELECT POWER(2, 3) AS result;`	*Raises 2 to the power of 3, yielding 8. Used in scientific, financial, or growth rate calculations.*

Examples:

Rounding Product Prices for Display:

```
SELECT product_name, ROUND(price, 1) AS rounded_price FROM Products;
```

Rounds price to one decimal place for a cleaner display in product listings.

Calculating Absolute Account Balances:

```
SELECT account_id, ABS(balance) AS positive_balance FROM Accounts;
```

Ensures all balance values are positive, useful when both positive and negative balances exist but need to be reported positively.

Using MOD to Identify Even Quantities:

```
SELECT item, quantity FROM Orders WHERE MOD(quantity, 2) = 0;
```

Retrieves items with even quantities, filtering rows where quantity divided by 2 has no remainder.

Calculating the Ceiling of Discounted Prices:

```
SELECT product_name, price, CEIL(price * 0.9) AS discounted_price FROM
Products;
```

Calculates a 10% discount on price and rounds up to the nearest integer for each product.

Calculating the Floor of Salary Increases:

```
SELECT employee_name, salary, FLOOR(salary * 1.05) AS new_salary FROM
Employees;
```

Calculates a 5% salary increase for each employee and rounds down to the nearest integer, ensuring no overestimation.

Using POWER to Calculate Exponential Growth:

```
SELECT POWER(1.05, 10) AS growth_factor;
```

Calculates an exponential growth factor over 10 periods at a 5% rate, often used in financial forecasting.

Joins

Join Type	Example	Description
INNER JOIN	`SELECT Orders.order_id, Customers.customer_name FROM Orders INNER JOIN Customers ON Orders.customer_id = Customers.customer_id;`	*Returns only rows where there is a match in both Orders and Customers based on customer_id.*
LEFT JOIN	`SELECT Customers.customer_name, Orders.order_id FROM Customers LEFT JOIN Orders ON Customers.customer_id = Orders.customer_id;`	*Returns all customers and their orders. If a customer has no order, order_id will be NULL.*
RIGHT JOIN	`SELECT Orders.order_id, Customers.customer_name FROM Orders RIGHT JOIN Customers ON Orders.customer_id = Customers.customer_id;`	*Returns all orders and their customer names. If an order has no customer, customer_name will be NULL.*
FULL JOIN	`SELECT Customers.customer_name, Orders.order_id FROM Customers FULL JOIN Orders ON Customers.customer_id = Orders.customer_id;`	*Returns all customers and all orders. Rows without matches in either table will display NULL values for the missing data.*
CROSS JOIN	`SELECT Products.product_name, Categories.category_name FROM Products CROSS JOIN Categories;`	*Returns every possible combination of Products and Categories, useful for generating all potential pairs.*
SELF JOIN	`SELECT A.employee_name AS employee, B.employee_name AS manager FROM Employees A JOIN Employees B ON A.manager_id = B.employee_id;`	*Lists employees along with their manager names by treating Employees as two separate tables (aliases A and B).*

Examples:

INNER JOIN for Matching Orders and Customers:

```
SELECT Orders.order_id, Customers.customer_name
FROM Orders
INNER JOIN Customers ON Orders.customer_id = Customers.customer_id;
```

Retrieves only orders with a matching customer in Customers.

LEFT JOIN to Include All Customers with or without Orders:

```
SELECT Customers.customer_name, Orders.order_id
FROM Customers
LEFT JOIN Orders ON Customers.customer_id = Orders.customer_id;
```

Returns all customers. For customers with no orders, order_id will be NULL.

RIGHT JOIN to List All Orders with or without Associated Customers:

```
SELECT Orders.order_id, Customers.customer_name
FROM Orders
RIGHT JOIN Customers ON Orders.customer_id = Customers.customer_id;
```

Returns all orders, including those that lack a corresponding customer entry.

FULL JOIN to List All Customers and Orders:

```
SELECT Customers.customer_name, Orders.order_id
FROM Customers
FULL JOIN Orders ON Customers.customer_id = Orders.customer_id;
```

Returns all customers and orders, with NULL for unmatched rows in either table.

CROSS JOIN:

```
SELECT Products.product_name, Categories.category_name
FROM Products
CROSS JOIN Categories;
```

Generates a complete pairing of each product with each category.

SELF JOIN to Display Employee-Manager Relationships:

```
SELECT A.employee_name AS employee, B.employee_name AS manager
FROM Employees A
JOIN Employees B ON A.manager_id = B.employee_id;
```

Lists each employee and their manager by joining Employees table to itself.

Set Operations

Operation	Example	Description
UNION	`SELECT city FROM Customers UNION SELECT city FROM Suppliers;`	*Combines results from two or more SELECT statements and removes duplicate rows by default. Both queries must have the same number and types of columns.*
UNION ALL	`SELECT city FROM Customers UNION ALL SELECT city FROM Suppliers;`	*Similar to UNION, but includes all rows, including duplicates.*
INTERSECT	`SELECT city FROM Customers INTERSECT SELECT city FROM Suppliers;`	*Returns only rows present in both result sets. Note that not all databases support INTERSECT.*
EXCEPT (or MINUS)	`SELECT city FROM Customers EXCEPT SELECT city FROM Suppliers;`	*Returns rows from the first query that do not appear in the second query's result set. Some databases (e.g., Oracle) use MINUS instead of EXCEPT.*

Examples:

UNION: Combining Unique Cities

```
SELECT city FROM Customers
UNION
SELECT city FROM Suppliers;
```

Retrieves a unique list of cities found in either Customers or Suppliers, removing duplicates.

UNION ALL: Combining All Cities with Duplicates

```
SELECT city FROM Customers
UNION ALL
SELECT city FROM Suppliers;
```

Combines cities from both tables and includes all occurrences, even if a city appears in both tables.

INTERSECT: Finding Common Cities

```
SELECT city FROM Customers
INTERSECT
SELECT city FROM Suppliers;
```

Returns only the cities that are present in both Customers and Suppliers tables. Not all databases support INTERSECT.

EXCEPT: Cities in Customers but not in Suppliers

```
SELECT city FROM Customers
EXCEPT
SELECT city FROM Suppliers;
```

Returns cities that exist in Customers but not in Suppliers. In Oracle databases, this operation would use MINUS instead of EXCEPT.

Subqueries

Subquery Type	Example	Description
Inline Subquery	`SELECT product_id, (SELECT AVG(price) FROM Products) AS avg_price FROM Products;`	*A subquery inside a SELECT statement to create calculated or aggregated columns.*
Subquery in WHERE	`SELECT name FROM Employees WHERE department_id IN (SELECT id FROM Departments WHERE location = 'New York');`	*A subquery used in the WHERE clause to filter results based on another query's result.*
Correlated Subquery	`SELECT e1.name FROM Employees e1 WHERE e1.salary > (SELECT AVG(e2.salary) FROM Employees e2 WHERE e1.department_id = e2.department_id);`	*A subquery that refers to columns from the outer query, evaluating the inner query for each row.*
EXISTS	`SELECT name FROM Employees WHERE EXISTS (SELECT * FROM Projects WHERE Projects.manager_id = Employees.id);`	*Checks if a subquery returns any results, often used to filter rows where a relationship exists.*
NOT EXISTS	`SELECT name FROM Employees WHERE NOT EXISTS (SELECT * FROM Projects WHERE Projects.manager_id = Employees.id);`	*Checks if a subquery returns no results, used to exclude rows without certain relationships.*

Examples:

Inline Subquery: Average Salary

```
SELECT name, salary, (SELECT AVG(salary) FROM Employees) AS avg_salary
FROM Employees;
```

Returns each employee's name, salary, and a column displaying the overall average salary from the Employees table.

Subquery in WHERE Clause: Employees in Specific Departments

```
SELECT name
FROM Employees
WHERE department_id IN (SELECT id FROM Departments WHERE name =
'Sales');
```

Finds employees who belong to the "Sales" department. The subquery retrieves the department IDs for departments named "Sales."

Correlated Subquery: Employees with Above-Average Salary in Their Department

```
SELECT e1.name
FROM Employees e1
WHERE e1.salary > (SELECT AVG(e2.salary) FROM Employees e2 WHERE
e1.department_id = e2.department_id);
```

Retrieves the names of employees who earn more than the average salary within their specific department. The subquery is correlated with the outer query based on department_id.

EXISTS: Employees Who Manage Projects

```
SELECT name
FROM Employees
WHERE EXISTS (SELECT 1 FROM Projects WHERE Projects.manager_id =
Employees.id);
```

Lists employees who manage at least one project. The EXISTS clause checks for any project with a manager_id matching an employee's id.

NOT EXISTS: Employees Without Managed Projects

```
SELECT name
FROM Employees
WHERE NOT EXISTS (SELECT 1 FROM Projects WHERE Projects.manager_id =
Employees.id);
```

Retrieves employees who do not manage any projects by using NOT EXISTS to exclude employees for whom the subquery returns a result.

Grouping and Aggregation

Function/Clause	Example	Description
GROUP BY	`SELECT department_id, COUNT(*) AS employee_count FROM Employees GROUP BY department_id;`	*Groups rows sharing a specified column value, often used with aggregate functions.*
HAVING	`SELECT department_id, COUNT(*) AS employee_count FROM Employees GROUP BY department_id HAVING employee_count > 10;`	*Filters groups based on conditions, often used with GROUP BY (similar to WHERE but for groups).*
Aggregating with GROUP BY	`SELECT department_id, AVG(salary) AS avg_salary FROM Employees GROUP BY department_id;`	*Uses aggregate functions with GROUP BY to perform calculations on each group.*
ROLLUP	`SELECT department_id, role, SUM(salary) AS total_salary FROM Employees GROUP BY ROLLUP(department_id, role);`	*Adds summary rows (subtotals, totals) to grouped results.*
CUBE	`SELECT department_id, role, SUM(salary) AS total_salary FROM Employees GROUP BY CUBE(department_id, role);`	*Produces subtotals for all combinations of grouping columns.*
GROUPING SETS	`SELECT department_id, role, SUM(salary) AS total_salary FROM Employees GROUP BY GROUPING SETS ((department_id), (role), (department_id, role));`	*Allows customized grouping combinations, useful for complex reports.*

Examples:

GROUP BY: Count Employees per Department

```
SELECT department_id, COUNT(*) AS employee_count
FROM Employees
GROUP BY department_id;
```

Groups employees by department and displays the count of employees in each.

HAVING Clause: Filter by Grouped Results

```
SELECT department_id, AVG(salary) AS avg_salary
FROM Employees
GROUP BY department_id
HAVING avg_salary > 50000;
```

Filters to only show departments where the average salary is above $50,000. HAVING applies the filter after grouping.

Aggregating with GROUP BY: Average and Sum

```
SELECT department_id, AVG(salary) AS avg_salary, SUM(salary) AS
total_salary
FROM Employees
GROUP BY department_id;
```

Returns each department's average and total salary, using GROUP BY to perform the calculations per department.

ROLLUP: Summaries at Multiple Levels

```
SELECT department_id, role, SUM(salary) AS total_salary
FROM Employees
GROUP BY ROLLUP(department_id, role);
```

Provides totals for each department-role combination, department-only totals, and a grand total. The ROLLUP clause creates subtotals and final totals.

CUBE: All Subtotal Combinations

```
SELECT department_id, role, SUM(salary) AS total_salary
FROM Employees
GROUP BY CUBE(department_id, role);
```

Creates totals for each possible combination of department_id and role, allowing for both individual subtotals and combinations of both fields.

GROUPING SETS: Custom Grouping Combinations

```
SELECT department_id, role, SUM(salary) AS total_salary
FROM Employees
GROUP BY GROUPING SETS ((department_id), (role), (department_id, role))
```

Allows customized grouping, calculating totals for each department, each role, and each department-role pair. This flexibility is helpful in generating complex reports.

Constraints

Constraint	Example	Description
PRIMARY KEY	`CREATE TABLE Employees (id INT PRIMARY KEY, name VARCHAR(100));`	*Uniquely identifies each record in a table. It cannot have NULL values and must be unique.*
FOREIGN KEY	`CREATE TABLE Orders (order_id INT PRIMARY KEY, employee_id INT, FOREIGN KEY (employee_id) REFERENCES Employees(id));`	*Creates a link between two tables by referencing the primary key of another table.*
UNIQUE	`CREATE TABLE Employees (email VARCHAR(100) UNIQUE, name VARCHAR(100));`	*Ensures that all values in a column are distinct, preventing duplicates.*
NOT NULL	`CREATE TABLE Employees (id INT NOT NULL, name VARCHAR(100));`	*Ensures that a column cannot have NULL values.*
CHECK	`CREATE TABLE Employees (id INT PRIMARY KEY, age INT CHECK (age >= 18));`	*Ensures that values in a column satisfy a given condition.*
DEFAULT	`CREATE TABLE Employees (id INT PRIMARY KEY, hire_date DATE DEFAULT CURRENT_DATE);`	*Sets a default value for a column when no value is provided during insertion.*
AUTO_INCREMENT	`CREATE TABLE Employees (id INT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(100));`	*Automatically generates a unique value for a primary key or unique column, typically used for IDs.*