```sql
SELECT DISTINCT Country FROM Customers;
```

```sql
SELECT COUNT(DISTINCT Country) FROM Customers;
```

OR

```sql
SELECT Count(*) AS DistinctCountries
FROM (SELECT DISTINCT Country FROM Customers);
```

```sql
SELECT * FROM Products
WHERE Price between 50 and 60;
```

```sql
SELECT * FROM Customers
WHERE City LIKE '%s%g';
```

```sql
SELECT * FROM Customers
WHERE City in ('Paris','London');
```

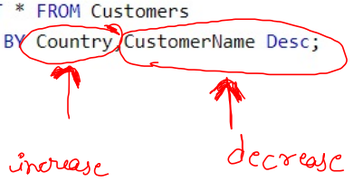
```sql
SELECT * FROM Products
WHERE Price != 18;

SELECT * FROM Products
WHERE Price <> 18;
```

```sql
SELECT * FROM Customers
WHERE Country='Germany' AND (City='Berlin' OR City='München');
```

```sql
SELECT * FROM Customers
WHERE NOT Country='Germany' AND NOT Country='USA';
```

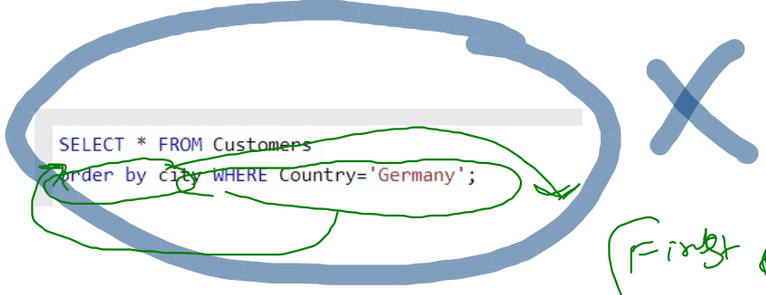
```sql
SELECT * FROM Customers
ORDER BY Country, CustomerName Desc;
```



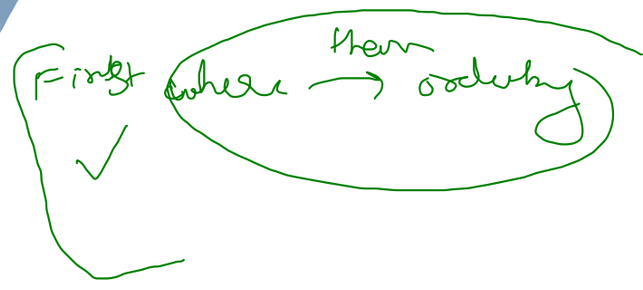

```sql
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

2. If you are adding values for all the columns of the table, you do n
names in the SQL query. However, make sure the order of the value
columns in the table. Here, the `INSERT INTO` syntax would be as fol

```sql
INSERT INTO table_name
VALUES (value1, value2, value3, ...);
```






## IS NULL Syntax

```sql
SELECT column_names
FROM table_name
WHERE column_name IS NULL;
```

## IS NOT NULL Syntax

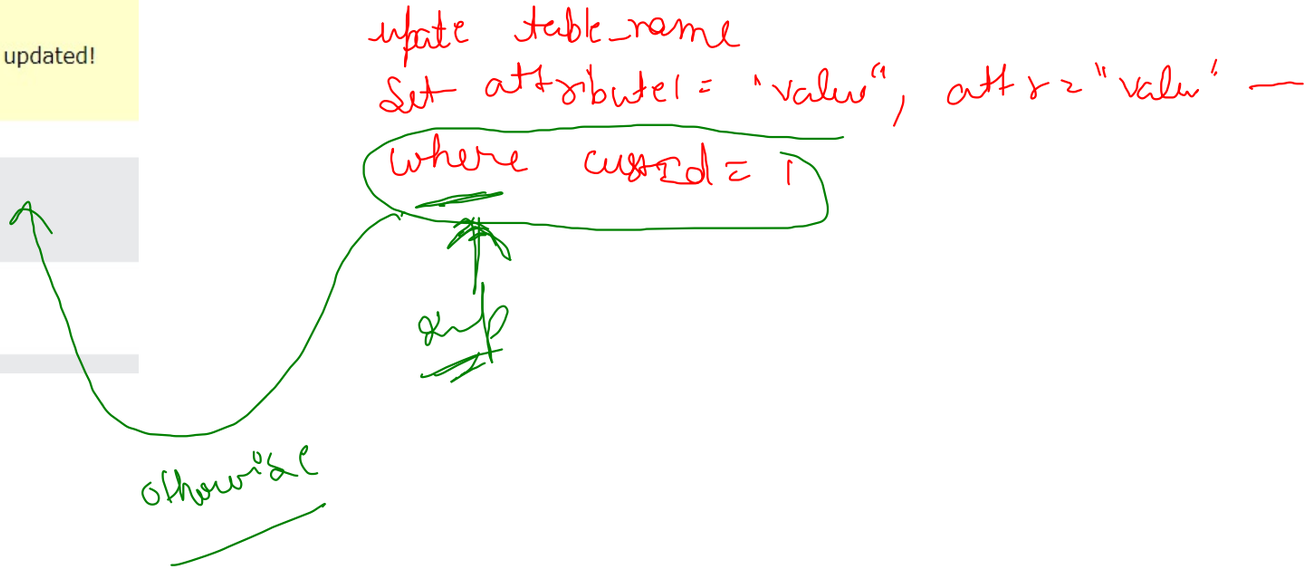
```sql
SELECT column_names
FROM table_name
WHERE column_name IS NOT NULL;
```

# Update Warning!

Be careful when updating records. If you omit the `WHERE` clause, ALL records will be updated!

## Example

```
UPDATE Customers
SET ContactName='Juan';
```



```sql
DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste';
```

# Delete All Records

It is possible to delete all rows in a table without deleting the table. This means that the table structure, attributes, and indexes will be intact:

```sql
DELETE FROM table_name;
```

**SQL Server / MS Access Syntax:**

```
SELECT TOP number|percent column_name(s)
FROM table_name
WHERE condition;
```

**MySQL Syntax:**

```
SELECT column_name(s)
FROM table_name
WHERE condition
LIMIT number;
```

**Oracle 12 Syntax:**

```
SELECT column_name(s)
FROM table_name
ORDER BY column_name(s)
FETCH FIRST number ROWS ONLY;
```

**Older Oracle Syntax:**

```
SELECT column_name(s)
FROM table_name
WHERE ROWNUM <= number;
```

**Older Oracle Syntax (with ORDER BY):**

```
SELECT *
FROM (SELECT column_name(s) FROM table_name ORDER BY column_name(s))
WHERE ROWNUM <= number;
```

# SQL TOP PERCENT Example

The following SQL statement selects the first 50% of the records from the "Customers" table (for SQL Server/MS Access):

## Example

```
SELECT TOP 50 PERCENT * FROM Customers;
```

Try it Yourself »

The following SQL statement shows the equivalent example for Oracle:

## Example

```
SELECT * FROM Customers
FETCH FIRST 50 PERCENT ROWS ONLY;
```

The following SQL statement selects the first three records from the "Custo
country is "Germany" (for SQL Server/MS Access):

## Example

```
SELECT TOP 3 * FROM Customers
WHERE Country='Germany';
```

Try it Yourself »

The following SQL statement shows the equivalent example for MySQL:

## Example

```
SELECT * FROM Customers
WHERE Country='Germany'
LIMIT 3;
```

The following SQL statement shows the equivalent example for Oracle:

## Example

```
SELECT * FROM Customers
WHERE Country='Germany'
FETCH FIRST 3 ROWS ONLY;
```

count(c_n), sum(c_n), avg(c_n), min(), max()

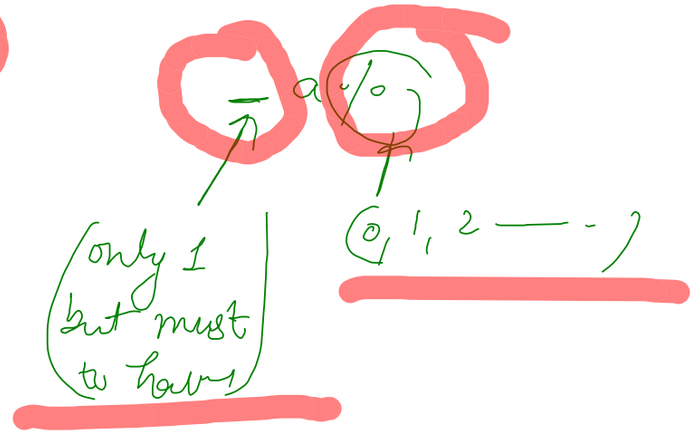
count(c_n), sum(c_n), avg(c_n), min(), max()

# LIKE

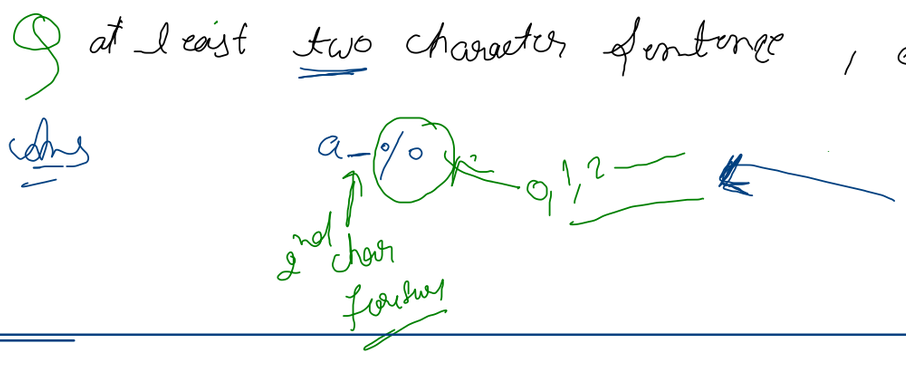
The `LIKE` operator is used in a `WHERE` clause to search for a specified pattern in a column.

There are two wildcards often used in conjunction with the `LIKE` operator:

- The percent sign (%) represents zero, one, or multiple characters
- The underscore sign (_) represents one, single character




Like




---

not like

```
SELECT * FROM Customers
WHERE CustomerName NOT LIKE 'a%';
```

not like

| Symbol | Description | Example |
|---|---|---|
| * | Represents zero or more characters | bl* finds bl, black, blue, and blob |
| ? | Represents a single character | h?t finds hot, hat, and hit |
| [] | Represents any single character within the brackets | h[oa]t finds hot and hat, but not hit |
| ! | Represents any character not in the brackets | h[!oa]t finds hit, but not hot and hat |
| - | Represents any single character within the specified range | c[a-b]t finds cat and cbt |
| # | Represents any single numeric character | 2#5 finds 205, 215, 225, 235, 245, 255, 265, 275, 285, and 295 |

```
SELECT * FROM Customers
WHERE City LIKE '[bsp]%';
```

The following SQL statement selects all customers with a City starting with "b", "s", or "p":

The following SQL statement selects all customers with a City starting with "a", "b", or "c":

## Example

```
SELECT * FROM Customers
WHERE City LIKE '[a-c]%';
```

The two following SQL statements select all customers with a City NOT starting with "b", "s", or "p"

## Example

```
SELECT * FROM Customers
WHERE City LIKE '[!bsp]%';
```

Try it Yourself »

Or:

## Example

```
SELECT * FROM Customers
WHERE City NOT LIKE '[bsp]%';
```

```sql
SELECT * FROM Products
WHERE Price NOT BETWEEN 10 AND 20;
```

```sql
SELECT * FROM Products
WHERE Price BETWEEN 10 AND 20
AND CategoryID NOT IN (1,2,3);
```

# SQL UNION Example

The following SQL statement returns the cities (only distinct values) from both the "Customers" and the "Suppliers" table:

## Example

```
SELECT City FROM Customers
UNION
SELECT City FROM Suppliers
ORDER BY City;
```

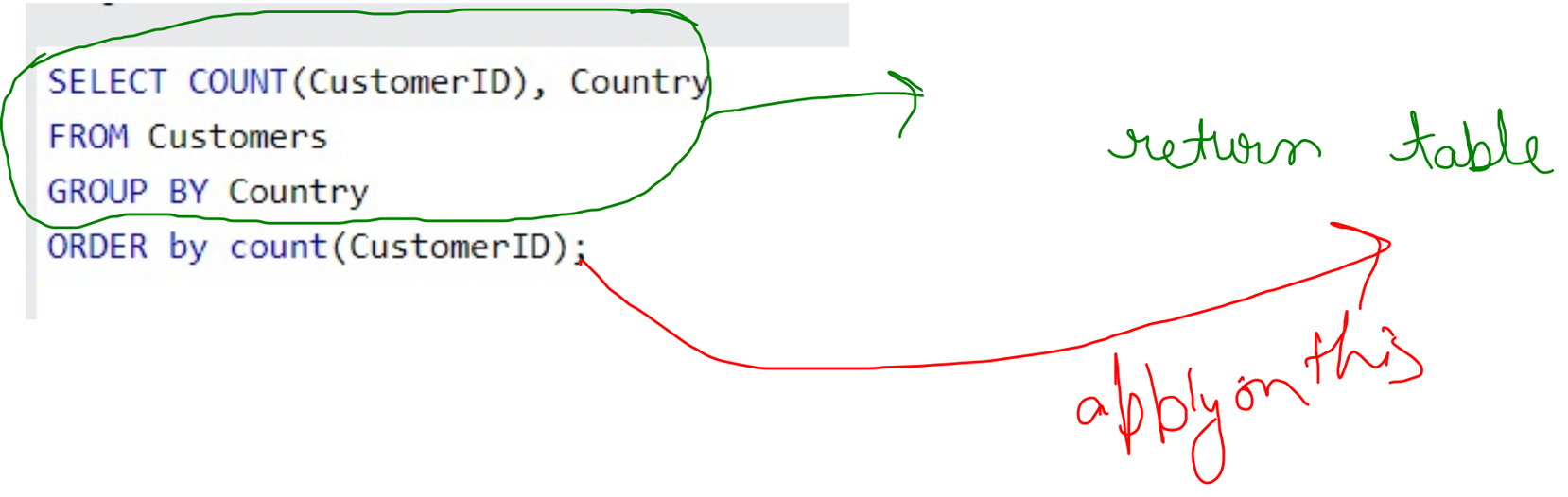
# SQL UNION ALL Example

The following SQL statement returns the cities (duplicate values also) from both the "Customers" and the "Suppliers" table:

## Example

```
SELECT City FROM Customers
UNION ALL
SELECT City FROM Suppliers
ORDER BY City;
```

```sql
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
ORDER by count(CustomerID);
```



# The SQL HAVING Clause

The `HAVING` clause was added to SQL because the `WHERE` keyword cannot be used with aggregate functions.

## HAVING Syntax

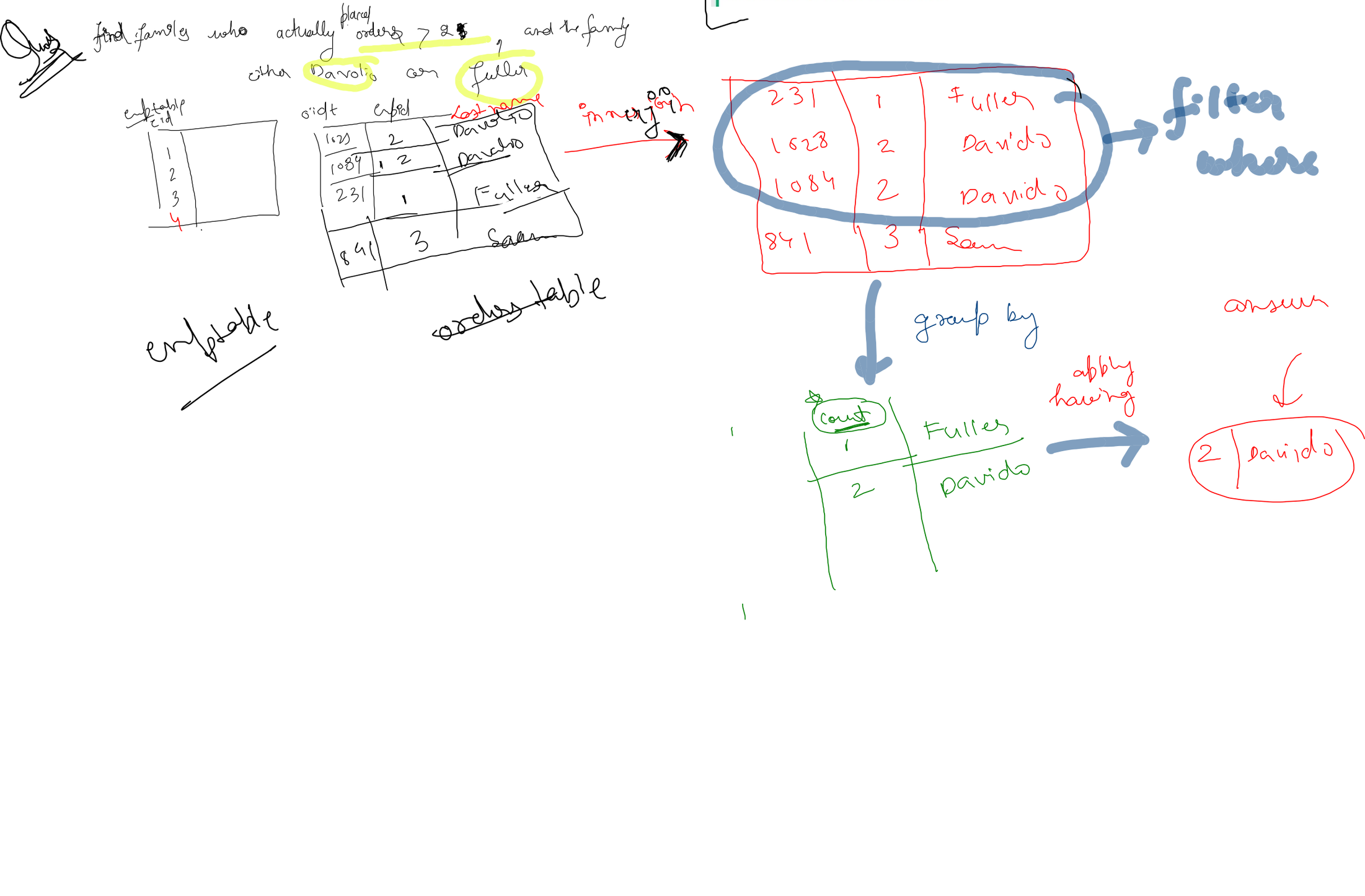
```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDER BY column_name(s);
```

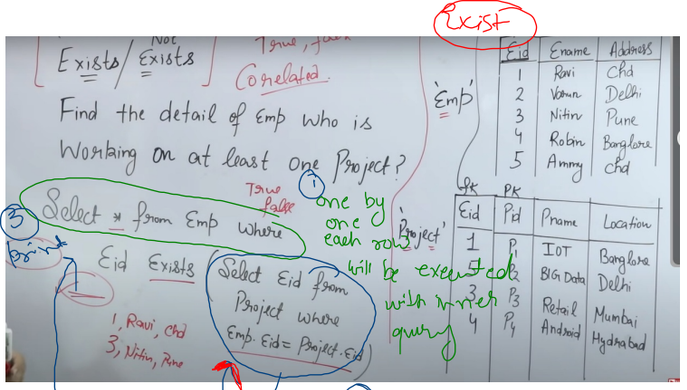
before
after

query

```sql
SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders
FROM Orders
INNER JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID
WHERE LastName = 'Davolio' OR LastName = 'Fuller'
GROUP BY LastName
HAVING COUNT(Orders.OrderID) > 25;
```

Ques find family who actually placed orders > 25, and the family either Davolio or Fuller

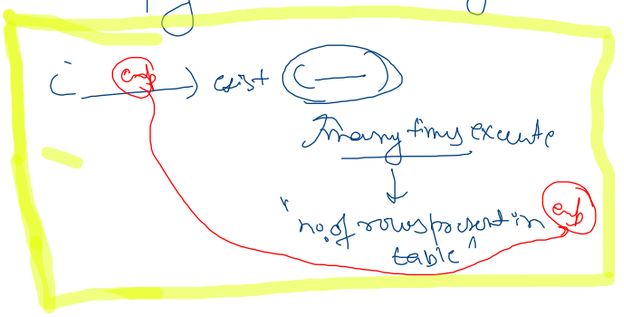





→ "correlated nested query"

→ follows top down approach

→ inner query is executed many times




loop for each row of emp (outer)

---

output:

| | | |
|---|---|---|
| 1 | Ravi | Chd |
| 3 | nitin | Pune |
| 4 | Robin | Banglore |
| 5 | Anny | Chd |

outer query ka table ke row utha ke jaa, or check karo inner table me exist or not

yes / no

true / false

nothing

value gets printed of emp

# Demo Database

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

| ProductID | ProductName | SupplierID | CategoryID | Unit | Price |
|---|---|---|---|---|---|
| 1 | Chais | 1 | 1 | 10 boxes x 20 bags | 18 |
| 2 | Chang | 1 | 1 | 24 - 12 oz bottles | 19 |
| 3 | Aniseed Syrup | 1 | 2 | 12 - 550 ml bottles | 10 |
| 4 | Chef Anton's Cajun Seasoning | 2 | 2 | 48 - 6 oz jars | 22 |
| 5 | Chef Anton's Gumbo Mix | 2 | 2 | 36 boxes | 21.35 |

And a selection from the "Suppliers" table:

| SupplierID | SupplierName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Exotic Liquid | Charlotte Cooper | 49 Gilbert St. | London | EC1 4SD | UK |
| 2 | New Orleans Cajun Delights | Shelley Burke | P.O. Box 78934 | New Orleans | 70117 | USA |
| 3 | Grandma Kelly's Homestead | Regina Murphy | 707 Oxford Rd. | Ann Arbor | 48104 | USA |
| 4 | Tokyo Traders | Yoshi Nagase | 9-8 Sekimai Musashino-shi | Tokyo | 100 | Japan |

The following SQL statement returns TRUE and lists the suppliers with a product price less than 20:

as we need Suppliers data, it must be in outer query

if any of the row in products matches this it just return true from there

```
select supplierId from suppliers where exists (select * from products where
suppliers.supplierID=products.supplierID and price<20);
```

Edit the SQL Statement, and click "Run SQL" to see the result.

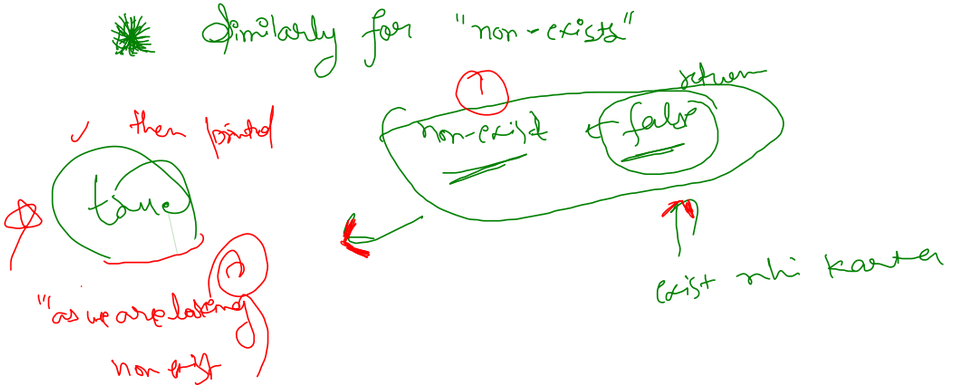
Run SQL »

Result:

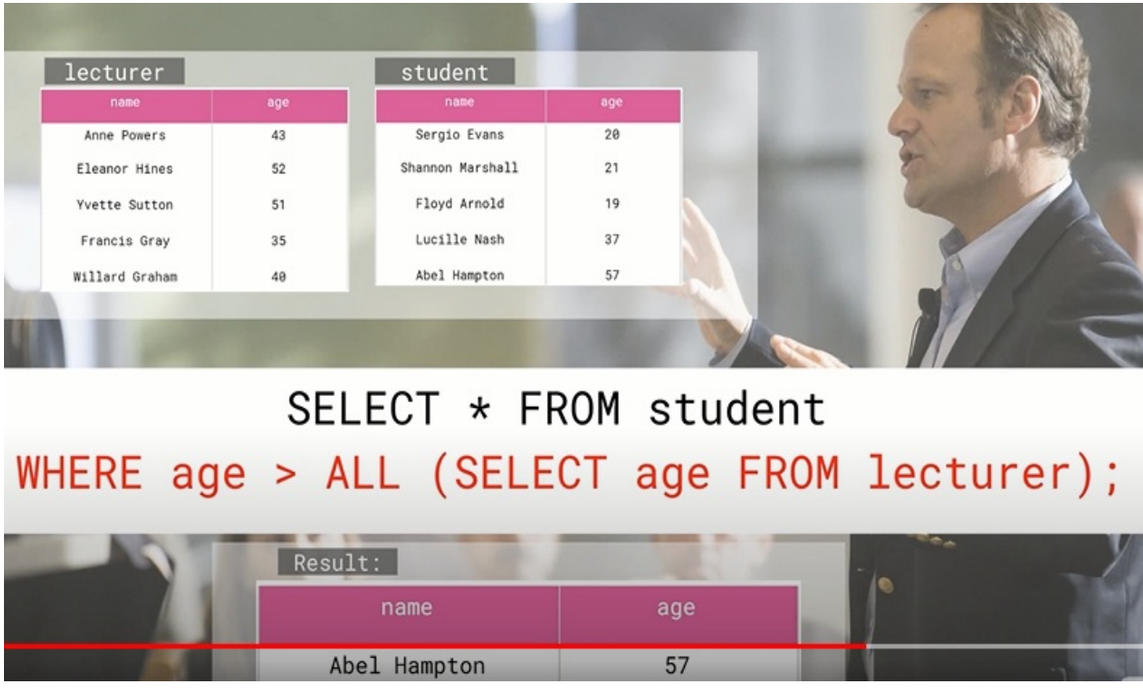
Number of Records: 24

| SupplierID |
|---|
| 1 |

Similarly for "non-exists"



- Select a Student whose age is greater than all lecturares.




x Point to be noted

→ there is no join operation →

→ no foreign key

lecturer

| name | age |
|---|---|
| Anne Powers | 43 |
| Eleanor Hines | 52 |
| Yvette Sutton | 51 |
| Francis Gray | 35 |
| Willard Graham | 40 |

student

| name | age |
|---|---|
| Sergio Evans | 20 |
| Shannon Marshall | 21 |
| Floyd Arnold | 19 |
| Lucille Nash | 37 |
| Abel Hampton | 57 |

The following SQL statement lists the ProductName if it finds ANY records in the OrderDetails table has Quantity larger than 1000 (this will return FALSE because the Quantity column has no values larger than 1000):

lecturer);

## Example

```
SELECT ProductName
FROM Products
WHERE ProductID = ANY
  (SELECT ProductID
  FROM OrderDetails
  WHERE Quantity > 1000);
```

Below is a selection from the **"Products"** table in the Northwind sample database:

| ProductID | ProductName | SupplierID | CategoryID | Unit | Price |
|---|---|---|---|---|---|
| 1 | Chais | 1 | 1 | 10 boxes x 20 bags | 18 |
| 2 | Chang | 1 | 1 | 24 - 12 oz bottles | 19 |
| 3 | Aniseed Syrup | 1 | 2 | 12 - 550 ml bottles | 10 |
| 4 | Chef Anton's Cajun Seasoning | 2 | 2 | 48 - 6 oz jars | 22 |
| 5 | Chef Anton's Gumbo Mix | 2 | 2 | 36 boxes | 21.35 |

And a selection from the **"OrderDetails"** table:

| OrderDetailID | OrderID | ProductID | Quantity |
|---|---|---|---|
| 1 | 10248 | 11 | 12 |
| 2 | 10248 | 42 | 10 |
| 3 | 10248 | 72 | 5 |
| 4 | 10249 | 14 | 9 |
| 5 | 10249 | 51 | 40 |
| 6 | 10250 | 41 | 10 |
| 7 | 10250 | 51 | 35 |
| 8 | 10250 | 65 | 15 |

The following SQL statement lists the ProductName if it finds ANY records in the OrderDetails table has Quantity equal to 10 (this will return TRUE because the Quantity column has some values of 10):

## Example

```
SELECT ProductName
FROM Products
WHERE ProductID = ANY
  (SELECT ProductID
  FROM OrderDetails
  WHERE Quantity = 10);
```

The following SQL statement lists the ProductName if it finds ANY records in the OrderDetails table has Quantity larger than 1000 (this will return FALSE because the Quantity column has no values larger than 1000):

## Example

```
SELECT ProductName
FROM Products
WHERE ProductID = ANY
  (SELECT ProductID
  FROM OrderDetails
  WHERE Quantity > 1000);
```

# SQL SELECT INTO Examples

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

The following SQL statement creates a backup copy of Customers:

```
SELECT * INTO CustomersBackup2017
FROM Customers;
```

newtable

The following SQL statement uses the `IN` clause to copy the table into a new table in another database:

```
SELECT * INTO CustomersBackup2017 IN 'Backup.mdb'
FROM Customers;
```

new database

The following SQL statement copies data from more than one table into a new table:

```
SELECT Customers.CustomerName, Orders.OrderID
INTO CustomersOrderBackup2017
FROM Customers
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

Colum instead of

table collected

**Tip:** `SELECT INTO` can also be used to create a new, empty table using the schema of another. Just add a `WHERE` clause that causes the query to return no data:

```
SELECT * INTO newtable
FROM oldtable
WHERE 1 = 0;
```

# The SQL INSERT INTO SELECT Statement

The `INSERT INTO SELECT` statement copies data from one table and inserts it into another table

The `INSERT INTO SELECT` statement requires that the data types in source and target tables match.

**Note:** The existing records in the target table are unaffected.

## INSERT INTO SELECT Syntax

Copy all columns from one table to another table:

```
INSERT INTO table2
SELECT * FROM table1
WHERE condition;
```

Copy only some columns from one table into another table:

```
INSERT INTO table2 (column1, column2, column3, ...)
SELECT column1, column2, column3, ...
FROM table1
WHERE condition;
```

```
INSERT INTO Customers (CustomerName, City, Country)
SELECT SupplierName, City, Country FROM Suppliers;
```

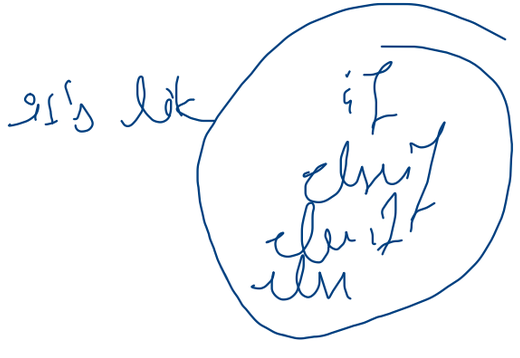
# CASE Syntax

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

## SQL Statement:

```
SELECT CustomerName,City,Country
FROM Customers
ORDER BY (CASE
    when city is null then country
    else city
END);
```

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



# SQL IFNULL(), ISNULL(), COALESCE(), and NVL() Functions

Look at the following "Products" table:

| P_Id | ProductName | UnitPrice | UnitsInStock | UnitsOnOrder |
|---|---|---|---|---|
| 1 | Jarlsberg | 10.45 | 16 | 15 |
| 2 | Mascarpone | 32.56 | 23 | |
| 3 | Gorgonzola | 15.67 | 9 | 20 |

**MS Access**

The MS Access `IsNull()` function returns TRUE (-1) if the expression is a null value, otherwise FALSE (0):

```
SELECT ProductName, UnitPrice * (UnitsInStock + IIF(IsNull(UnitsOnOrder), 0, UnitsOnOrder))
FROM Products;
```

**Oracle**

The Oracle `NVL()` function achieves the same result:

```
SELECT ProductName, UnitPrice * (UnitsInStock + NVL(UnitsOnOrder, 0))
FROM Products;
```

## Solutions

**MySQL**

The MySQL `IFNULL()` function lets you return an alternative value if an expression is NULL:

```
SELECT ProductName, UnitPrice * (UnitsInStock + IFNULL(UnitsOnOrder, 0))
FROM Products;
```

or we can use the `COALESCE()` function, like this:

```
SELECT ProductName, UnitPrice * (UnitsInStock + COALESCE(UnitsOnOrder, 0))
FROM Products;
```

**SQL Server**

The SQL Server `ISNULL()` function lets you return an alternative value when an expression is NULL:

```
SELECT ProductName, UnitPrice * (UnitsInStock + ISNULL(UnitsOnOrder, 0))
FROM Products;
```

# What is a Stored Procedure?

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.

You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

## Stored Procedure Syntax

```
CREATE PROCEDURE procedure_name
AS
sql_statement
GO;
```

## Execute a Stored Procedure

```
EXEC procedure_name;
```

# Stored Procedure With Multiple Parameters

Setting up multiple parameters is very easy. Just list each parameter and the data type separated by a comma as shown below.

The following SQL statement creates a stored procedure that selects Customers from a particular City with a particular PostalCode from the "Customers" table:

## Example

```
CREATE PROCEDURE SelectAllCustomers @City nvarchar(30), @PostalCode nvarchar(10)
AS
SELECT * FROM Customers WHERE City = @City AND PostalCode = @PostalCode
GO;
```

Execute the stored procedure above as follows:

## Example

```
EXEC SelectAllCustomers @City = 'London', @PostalCode = 'WA1 1DP';
```

# Comments

The following example uses a multi-line comment to ignore many statements:

## Example

```
/*SELECT * FROM Customers;
SELECT * FROM Products;
SELECT * FROM Orders;
SELECT * FROM Categories;*/
SELECT * FROM Suppliers;
```

## Example

```
--SELECT * FROM Customers;
SELECT * FROM Products;
```

Try it Yourself »

→ Single line