The SQL SELECT DISTINCT Statement

The `SELECT DISTINCT` statement is used to return only distinct (different) values.

Example

Select all the different countries from the "Customers" table:

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

Count Distinct

By using the `DISTINCT` keyword in a function called `COUNT`, we can return the number of different countries.

Example

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

The SQL WHERE Clause

The `WHERE` clause is used to filter records.

It is used to extract only those records that fulfill a specified condition.

Example

Get your own SQL Server

Select all customers from Mexico:

```
SELECT * FROM Customers

WHERE Country='Mexico';
```

Operators in The WHERE Clause

You can use other operators than the = operator to filter the search.

Example

Select all customers with a CustomerID greater than 80:

```
SELECT * FROM Customers

WHERE CustomerID > 80;
```

The following operators can be used in the `WHERE` clause:

Operator	Description
=	Equal
>	Greater than
<	Less than

>=	Greater than or equal
<=	Less than or equal
<>	Not equal. Note: In some versions of SQL this operator may be written as !=
BETWEEN	Between a certain range
LIKE	Search for a pattern
IN	To specify multiple possible values for a column

The SQL ORDER BY

The `ORDER BY` keyword is used to sort the result-set in ascending or descending order.

Example

Sort the products by price:

```
SELECT * FROM Products

ORDER BY Price;
```

Syntax

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1, column2, ... ASC|DESC;
```

DESC

The `ORDER BY` keyword sorts the records in ascending order by default. To sort the records in descending order, use the `DESC` keyword.

Example

Sort the products from highest to lowest price:

```
SELECT * FROM Products

ORDER BY Price DESC;
```

Using Both ASC and DESC

The following SQL statement selects all customers from the "Customers" table, sorted ascending by the "Country" and descending by the "CustomerName" column:

Example

```
SELECT * FROM Customers
```

```
ORDER BY Country ASC, CustomerName DESC;
```

The SQL LIKE Operator

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

You will learn more about wildcards in the next chapter.

Example

Select all customers that starts with the letter "a":

```
SELECT * FROM Customers

WHERE CustomerName LIKE 'a%';
```

The _ Wildcard

The `_` wildcard represents a single character.

It can be any character or number, but each `_` represents one, and only one, character.

Example

Return all customers from a city that starts with 'L' followed by one wildcard character, then 'nd' and then two wildcard characters:

```
SELECT * FROM Customers

WHERE city LIKE 'L_nd__';
```

The SQL AND Operator

The `WHERE` clause can contain one or many `AND` operators.

The `AND` operator is used to filter records based on more than one condition, like if you want to return all customers from Spain that starts with the letter 'G':

Example

Select all customers from Spain that starts with the letter 'G':

```
SELECT *

FROM Customers

WHERE Country = 'Spain' AND CustomerName LIKE 'G%';
```

The SQL OR Operator

The `WHERE` clause can contain one or more `OR` operators.

The `OR` operator is used to filter records based on more than one condition, like if you want to return all customers from Germany but also those from Spain:

Example

Select all customers from Germany or Spain:

```
SELECT *

FROM Customers

WHERE Country = 'Germany' OR Country = 'Spain';
```

Combining AND and OR

You can combine the `AND` and `OR` operators.

The following SQL statement selects all customers from Spain that starts with a "G" or an "R".

Make sure you use parenthesis to get the correct result.

Example

Select all Spanish customers that starts with either "G" or "R":

```
SELECT * FROM Customers

WHERE Country = 'Spain' AND (CustomerName LIKE 'G%' OR CustomerName
LIKE 'R%');
```

The NOT Operator

The `NOT` operator is used in combination with other operators to give the opposite result, also called the negative result.

In the select statement below we want to return all customers that are NOT from Spain:

Example

Select only the customers that are NOT from Spain:

```
SELECT * FROM Customers

WHERE NOT Country = 'Spain';
```

NOT LIKE

Example

Select customers that does not start with the letter 'A':

```
SELECT * FROM Customers

WHERE CustomerName NOT LIKE 'A%';
```

NOT BETWEEN

Example

Select customers with a customerID not between 10 and 60:

```
SELECT * FROM Customers

WHERE CustomerID NOT BETWEEN 10 AND 60;
```

NOT IN

Example

Select customers that are not from Paris or London:

```
SELECT * FROM Customers

WHERE City NOT IN ('Paris', 'London');
```

NOT Greater Than

Example

Select customers with a CustomerId not greater than 50:

```
SELECT * FROM Customers

WHERE NOT CustomerID > 50;
```

Note: There is a not-greater-then operator: `!>` that would give you the same result.

NOT Less Than

Example

Select customers with a CustomerID not less than 50:

```
SELECT * FROM Customers

WHERE NOT CustomerId < 50;
```

The SQL INSERT INTO Statement

The `INSERT INTO` statement is used to insert new records in a table.

INSERT INTO Syntax

It is possible to write the `INSERT INTO` statement in two ways:

1. Specify both the column names and the values to be inserted:

```
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 not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table. Here, the `INSERT INTO` syntax would be as follows:

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

Insert Multiple Rows

It is also possible to insert multiple rows in one statement.

To insert multiple rows of data, we use the same `INSERT INTO` statement, but with multiple values:

Example

```
INSERT INTO Customers (CustomerName, ContactName, Address, City,
PostalCode, Country)


VALUES
```

```
('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006',
'Norway'),


('Greasy Burger', 'Per Olsen', 'Gateveien 15', 'Sandnes', '4306',
'Norway'),


('Tasty Tee', 'Finn Egan', 'Streetroad 19B', 'Liverpool', 'L1 0AA',
'UK');
```

Make sure you separate each set of values with a comma `,`.

What is a NULL Value?

A field with a NULL value is a field with no value.

If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value.

Note: A NULL value is different from a zero value or a field that contains spaces. A field with a NULL value is one that has been left blank during record creation!

How to Test for NULL Values?

It is not possible to test for NULL values with comparison operators, such as =, <, or <>.

We will have to use the `IS NULL` and `IS NOT NULL` operators instead.

IS NULL Syntax

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

IS NOT NULL Syntax

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

The IS NULL Operator

The `IS NULL` operator is used to test for empty values (NULL values).

The following SQL lists all customers with a NULL value in the "Address" field:

Example

```
SELECT CustomerName, ContactName, Address

FROM Customers

WHERE Address IS NULL;
```

Tip: Always use IS NULL to look for NULL values.

The IS NOT NULL Operator

The `IS NOT NULL` operator is used to test for non-empty values (NOT NULL values).

The following SQL lists all customers with a value in the "Address" field:

Example

```
SELECT CustomerName, ContactName, Address

FROM Customers

WHERE Address IS NOT NULL;
```

The SQL UPDATE Statement

The `UPDATE` statement is used to modify the existing records in a table.

UPDATE Syntax

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

Note: Be careful when updating records in a table! Notice the `WHERE` clause in the `UPDATE` statement. The `WHERE` clause specifies which record(s) that should be updated. If you omit the `WHERE` clause, all records in the table will be updated!

UPDATE Table

The following SQL statement updates the first customer (CustomerID = 1) with a new contact person *and* a new city.

Example

```
UPDATE Customers
```

```
SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'

WHERE CustomerID = 1;
```

The SQL DELETE Statement

The `DELETE` statement is used to delete existing records in a table.

DELETE Syntax

```
DELETE FROM table_name WHERE condition;
```

Note: Be careful when deleting records in a table! Notice the `WHERE` clause in the `DELETE` statement. The `WHERE` clause specifies which record(s) should be deleted. If you omit the `WHERE` clause, all records in the table will be deleted!

SQL DELETE Example

The following SQL statement deletes the customer "Alfreds Futterkiste" from the "Customers" table:

Example

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

The SQL MIN() and MAX() Functions

The MIN() function returns the smallest value of the selected column.

The MAX() function returns the largest value of the selected column.

MIN Example

Find the lowest price:

```
SELECT MIN(Price)

FROM Products;
```

MAX Example

Find the highest price:

```
SELECT MAX(Price)

FROM Products;
```

Set Column Name (Alias)

When you use `MIN()` or `MAX()`, the returned column will be named `MIN(field)` or `MAX(field)` by default. To give the column a new name, use the `AS` keyword:

Example

```
SELECT MIN(Price) AS SmallestPrice

FROM Products;
```

The SQL COUNT() Function

The `COUNT()` function returns the number of rows that matches a specified criterion.

Example

Find the total number of products in the `Products` table:

```
SELECT COUNT(*)

FROM Products;
```

The SQL SUM() Function

The `SUM()` function returns the total sum of a numeric column.

Example

Return the sum of all `Quantity` fields in the `OrderDetails` table:

```
SELECT SUM(Quantity)

FROM OrderDetails;
```

Syntax

```
SELECT SUM(column_name)
FROM table_name
WHERE condition;
```

Use an Alias

Give the summarized column a name by using the `AS` keyword.

Example

Name the column "total":

```
SELECT SUM(Quantity) AS total

FROM OrderDetails;
```

SUM() With an Expression

The parameter inside the `SUM()` function can also be an expression.

If we assume that each product in the `OrderDetails` column costs 10 dollars, we can find the total earnings in dollars by multiply each quantity with 10:

Example

Use an expression inside the `SUM()` function:

```
SELECT SUM(Quantity * 10)

FROM OrderDetails;
```

The SQL AVG() Function

The `AVG()` function returns the average value of a numeric column.

Example

Find the average price of all products:

```
SELECT AVG(Price)

FROM Products;
```

Note: NULL values are ignored.

Syntax

```
SELECT AVG(column_name)
FROM table_name
WHERE condition;
```

The 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.

Example

Return all customers from 'Germany', 'France', or 'UK'

```
SELECT * FROM Customers
```

```
WHERE Country IN ('Germany', 'France', 'UK');
```

Syntax

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (value1, value2, ...);
```

NOT IN

By using the `NOT` keyword in front of the `IN` operator, you return all records that are NOT any of the values in the list.

Example

Return all customers that are NOT from 'Germany', 'France', or 'UK':

```
SELECT * FROM Customers

WHERE Country NOT IN ('Germany', 'France', 'UK');
```

IN (SELECT)

You can also use `IN` with a subquery in the `WHERE` clause.

With a subquery you can return all records from the main query that are present in the result of the subquery.

Example

Return all customers that have an order in the Orders table:

```
SELECT * FROM Customers

WHERE CustomerID IN (SELECT CustomerID FROM Orders);
```

NOT IN (SELECT)

The result in the example above returned 74 records, that means that there are 17 customers that haven't placed any orders.

Let us check if that is correct, by using the `NOT IN` operator.

Example

Return all customers that have NOT placed any orders in the Orders table:

```
SELECT * FROM Customers

WHERE CustomerID NOT IN (SELECT CustomerID FROM Orders);
```

The SQL BETWEEN Operator

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.

Example

Selects all products with a price between 10 and 20:

```
SELECT * FROM Products

WHERE Price BETWEEN 10 AND 20;
```

Syntax

```
SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

NOT BETWEEN

To display the products outside the range of the previous example, use `NOT BETWEEN`:

Example

```
SELECT * FROM Products

WHERE Price NOT BETWEEN 10 AND 20;
```

BETWEEN with IN

The following SQL statement selects all products with a price between 10 and 20. In addition, the CategoryID must be either 1,2, or 3:

Example

```
SELECT * FROM Products

WHERE Price BETWEEN 10 AND 20

AND CategoryID IN (1,2,3);
```

BETWEEN Dates

The following SQL statement selects all orders with an OrderDate between '01-July-1996' and '31-July-1996':

Example

```
SELECT * FROM Orders

WHERE OrderDate BETWEEN '1996-07-01' AND '1996-07-31';
```