

## Laboratory work

Topic: Getting started with Microsoft SQL Server. Basics of T-SQL. DML.

Purpose: Adding data. INSERT command. Fetching data. SELECT command

Sorting. ORDER BY. Extracting a range of rows. Filtering. WHERE Filter operators. Refresh data. UPDATE command Delete data. DELETE command.

### Progress

#### Adding data. INSERT command

To add data, the INSERT command is used, which has the following formal syntax:

```
1  INSERT [INTO] table_name [(columns_list)] VALUES (value1, value2, ... valueN)
```

At the beginning there is an INSERT INTO statement, then in brackets you can specify a list of columns separated by commas to which data should be added, and at the end, after the word VALUES, in brackets, the values to be added for the columns are listed.

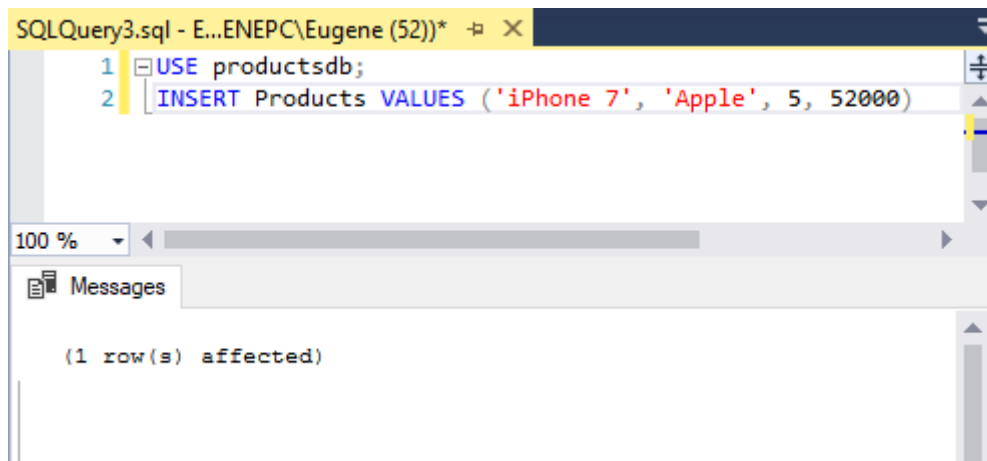
For example, let's say the following database was created earlier:

```
1  CREATE DATABASE productsdb;
2  GO
3  USE productsdb;
4  CREATE TABLE Products
5  (
6  Id INT IDENTITY PRIMARY KEY,
7  ProductName NVARCHAR(30) NOT NULL,
8  NVARCHAR(20) NOT NULL,
9  ProductCount INT DEFAULT 0,
10 Price MONEY NOT NULL
11 )
```

Let's add one line to it using the INSERT command:

one INSERT Products VALUES ('iPhone 7', 'Apple', 5, 52000)

After successful execution in SQL Server Management Studio, the message "1 row(s) affected" should appear in the message box:



Note that the values for the columns in parentheses after the VALUES keyword are passed in the order in which they are declared. For example, in the CREATE TABLE statement above, you can see that the first column is Id. But since the IDENTITY attribute is set for it, the value of this column is automatically generated, and it can be omitted. The second column represents ProductName, so the first value, the string "iPhone 7", will be passed to that column. The second value, the string "Apple", will be passed to the third column Manufacturer, and so on. That is, the values are passed to the columns as follows:

- ProductName: 'iPhone 7'
- Manufacturer: 'Apple'
- ProductCount: 5
- Price: 52000

Also, when entering values, you can specify the immediate columns in which values will be added:

```
1  INSERT INTO Products (ProductName, Price, Manufacturer)
2  VALUES('iPhone 6S', 41000, 'Apple')
```

Here the value is specified for only three columns. And now the values are passed in the order of the columns:

- ProductName: 'iPhone 6S'
- Manufacturer: 'Apple'
- Price: 41000

For unspecified columns (in this case, ProductCount), a default value will be added if the DEFAULT attribute is set, or NULL. However, unspecified columns must be nullable or have a DEFAULT attribute.

We can also add multiple lines at once:

```
1  INSERT INTO Products
2  VALUES
3  ('iPhone 6', 'Apple', 3, 36000),
4  ('Galaxy S8', 'Samsung', 2, 46000),
5  ('Galaxy S8 Plus', 'Samsung', 1, 56000)
```

In this case, three rows will be added to the table.

Also, when adding, we can specify that the default value is used for the column using the DEFAULT keyword or NULL:

```
1  INSERT INTO Products (ProductName, Manufacturer, ProductCount, Price)
2  VALUES('Mi6', 'Xiaomi', DEFAULT, 28000)
```

In this case, the default value will be used for the ProductCount column (if it is set, if not, then NULL).

If all columns have a DEFAULT attribute that defines a default value, or are nullable, then you can insert default values for all columns:

```
1  INSERT INTO Products
2  DEFAULT VALUES
```

But if you take the Products table, then such a command will fail, since several fields do not have a DEFAULT attribute and at the same time do not allow null values.

## **Data sampling. SELECT command**

The SELECT command is used to retrieve data. In simplified form, it has the following syntax:

```
1  SELECT column_list FROM table_name
```

For example, let's say the Products table was created earlier and some initial data was added to it:

```
1  CREATE TABLE Products
2  (
3  Id INT IDENTITY PRIMARY KEY,
4  ProductName NVARCHAR(30) NOT NULL,
5  NVARCHAR(20) NOT NULL,
```

```

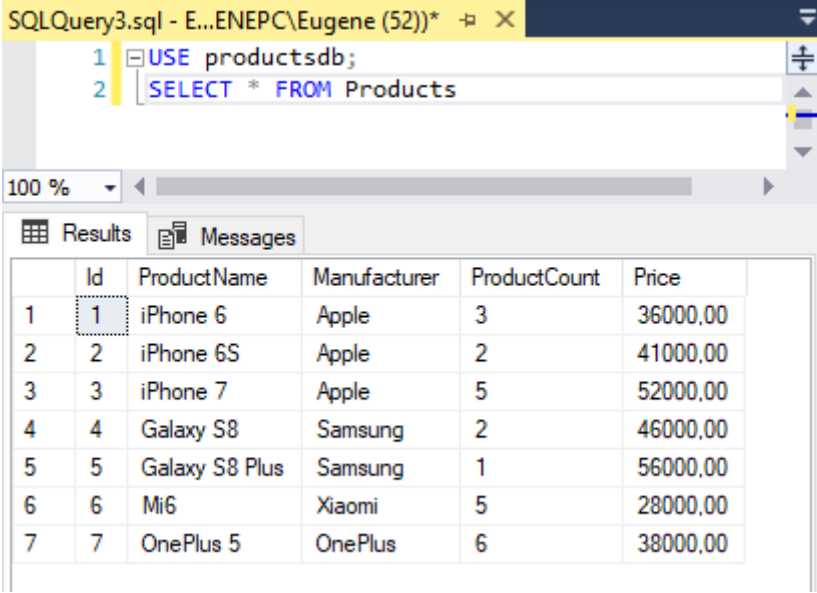
6    ProductCount INT DEFAULT 0,
7    Price MONEY NOT NULL
8    );
9
10   INSERT INTO Products
11   VALUES
12   ('iPhone 6', 'Apple', 3, 36000),
13   ('iPhone 6S', 'Apple', 2, 41000),
14   ('iPhone 7', 'Apple', 5, 52000),
15   ('Galaxy S8', 'Samsung', 2, 46000),
16   ('Galaxy S8 Plus', 'Samsung', 1, 56000),
17   ('Mi6', 'Xiaomi', 5, 28000),
18   ('OnePlus 5', 'OnePlus', 6, 38000)

```

Let's get all objects from this table:

```
1  SELECT * FROM Products
```

The asterisk \* indicates that we need to get all the columns.



The screenshot shows a SQL query window titled 'SQLQuery3.sql - E...ENEPC\Eugene (52))' containing the following SQL code:

```

1  USE productsdb;
2  SELECT * FROM Products

```

Below the query window, the 'Results' tab is active, displaying a table with the following data:

	Id	ProductName	Manufacturer	ProductCount	Price
1	1	iPhone 6	Apple	3	36000,00
2	2	iPhone 6S	Apple	2	41000,00
3	3	iPhone 7	Apple	5	52000,00
4	4	Galaxy S8	Samsung	2	46000,00
5	5	Galaxy S8 Plus	Samsung	1	56000,00
6	6	Mi6	Xiaomi	5	28000,00
7	7	OnePlus 5	OnePlus	6	38000,00

Retrieving all columns with an asterisk \* is considered not a good practice, as usually not all columns are needed. And a better approach is to include all required columns after the SELECT word. The exception is the case when you need to get data on absolutely all columns of the table. Also, using the \* character may be preferable in situations where the exact names of the columns are not known.

If we need to get data not for all, but for some specific columns, then all these column specifications are listed with a comma after SELECT:

```
1  SELECT ProductName, Price FROM Products
```

SQLQuery3.sql - E...ENEPC\Eugene (52))\*

```

1 USE productsdb;
2 SELECT ProductName, Price FROM Products

```

100 %

Results Messages

	ProductName	Price
1	iPhone 6	36000,00
2	iPhone 6S	41000,00
3	iPhone 7	52000,00
4	Galaxy S8	46000,00
5	Galaxy S8 Plus	56000,00
6	Mi6	28000,00
7	OnePlus 5	38000,00

A column specification does not have to represent its name. It can be any expression, such as the result of an arithmetic operation. So let's run the following query:

```

1 SELECT ProductName + ' (' + Manufacturer + ')', Price, Price * ProductCount
2 FROM Products

```

Here, the selection will create three columns. The first column represents the result of combining the two columns ProductName and Manufacturer. The second column is the standard Price column. And the third column represents the value of the Price column multiplied by the value of the ProductCount column.

SQLQuery3.sql - E...ENEPC\Eugene (52))\* SQLQuery2.sql - E...ENEPC\Eugene (51))\*

```

1 USE productsdb;
2
3 SELECT ProductName + ' (' + Manufacturer + ')', Price, Price * ProductCount
4 FROM Products

```

100 %

Results Messages

	(No column name)	Price	(No column name)
1	iPhone 6 (Apple)	36000,00	108000,00
2	iPhone 6S (Apple)	41000,00	82000,00
3	iPhone 7 (Apple)	52000,00	260000,00
4	Galaxy S8 (Samsung)	46000,00	92000,00
5	Galaxy S8 Plus (Samsung)	56000,00	56000,00
6	Mi6 (Xiaomi)	28000,00	140000,00
7	OnePlus 5 (OnePlus)	38000,00	228000,00

With the AS operator, you can change the name of the output column or define its alias:

```

1 SELECT

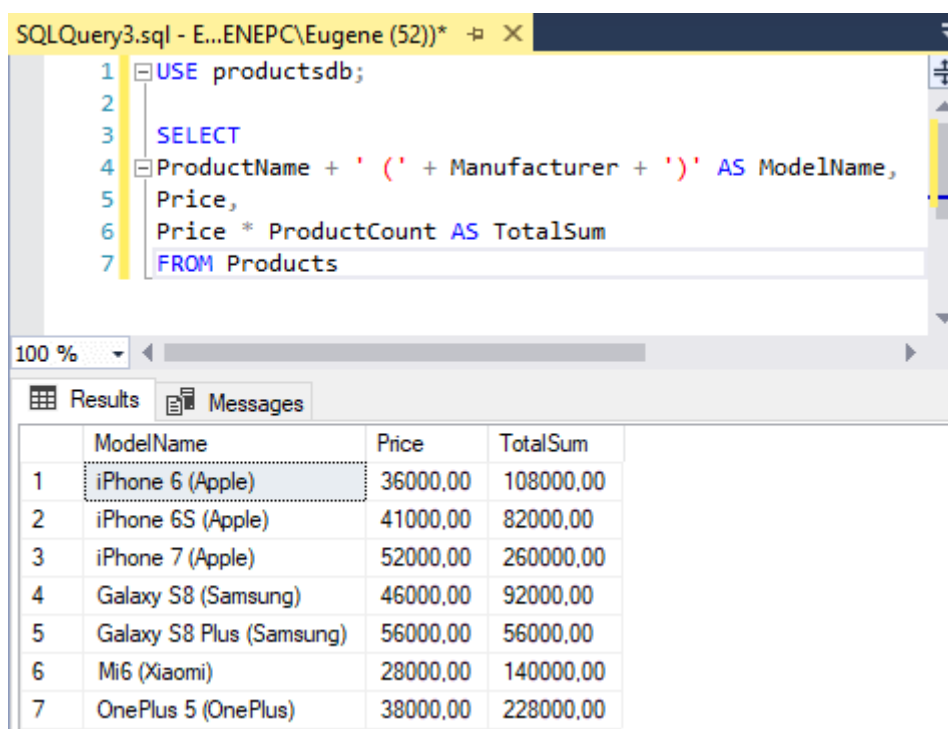
```

```

2  ProductName + ' (' + Manufacturer + ')' AS ModelName,
3  price,
4  Price * ProductCount AS TotalSum
5  FROM Products

```

In this case, the result of the selection is data on 3 columns. The first column, ModelName, combines the ProductName and Manufacturere columns, and the second represents the standard Price column. The third column, TotalSum, stores the product of the ProductCount and Price columns. In this case, as in the case of the Price column, it is not necessary to determine the name of the resulting column using AS.



The screenshot shows a SQL query window titled 'SQLQuery3.sql - E...ENEPC\Eugene (52)\*'. The query is as follows:

```

1  USE productsdb;
2
3  SELECT
4  ProductName + ' (' + Manufacturer + ')' AS ModelName,
5  Price,
6  Price * ProductCount AS TotalSum
7  FROM Products

```

Below the query window, the 'Results' tab is active, displaying a table with 7 rows and 4 columns: 'ModelName', 'Price', and 'TotalSum'. The first column is implicitly 'id'.

	ModelName	Price	TotalSum
1	iPhone 6 (Apple)	36000,00	108000,00
2	iPhone 6S (Apple)	41000,00	82000,00
3	iPhone 7 (Apple)	52000,00	260000,00
4	Galaxy S8 (Samsung)	46000,00	92000,00
5	Galaxy S8 Plus (Samsung)	56000,00	56000,00
6	Mi6 (Xiaomi)	28000,00	140000,00
7	OnePlus 5 (OnePlus)	38000,00	228000,00

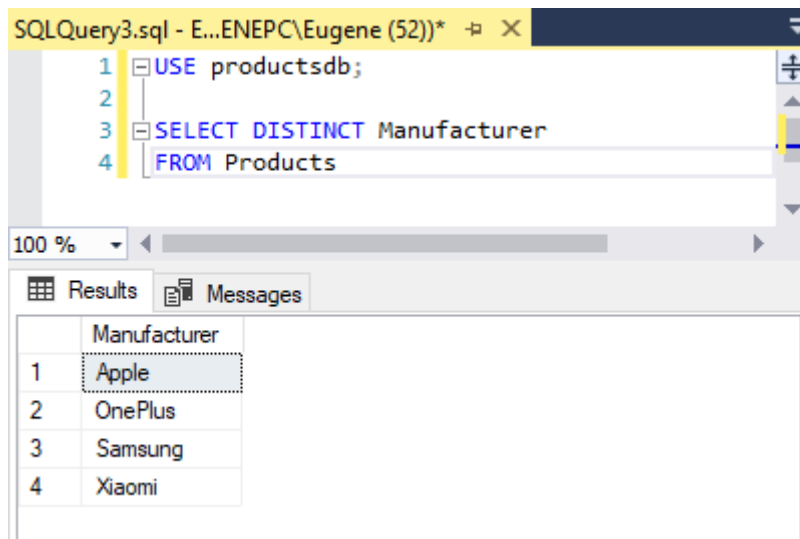
## DISTINCT

The DISTINCT statement allows you to select unique rows. For example, in our case, the table may contain several products from the same manufacturers. Let's select all manufacturers:

```

1  SELECT DISTINCT manufacturer
2  FROM Products

```



In this case, the row delimiter is the Manufacturer column. Therefore, the resulting selection will contain only unique Manufacturer values. And if, for example, in the database there are two products with the manufacturer Apple, then this name will occur in the resulting selection only once.

Sample with addition

### *SELECT INTO*

The **SELECT INTO** statement allows you to select some data from one table to another table, while the second table is created automatically. For example:

```
1  SELECT ProductName + '(' + Manufacturer + ')' AS ModelName, Price
2  INTO ProductSummary
3  FROM Products
4
5  SELECT * FROM ProductSummary
```

After executing this command, another ProductSummary table will be created in the database, which will have two columns ModelName and Price, and the data for these columns will be taken from the Products table:

SQLQuery3.sql - E...ENEPC\Eugene (52))\* SQLQuery2.sql - E...ENEPC\Eugene (51))\*

```

1 USE productsdb;
2
3 SELECT ProductName + ' (' + Manufacturer + ')' AS ModelName, Price
4 INTO ProductSummary
5 FROM Products
6
7 SELECT * FROM ProductSummary

```

100 %

Results Messages

	ModelName	Price
1	iPhone 6 (Apple)	36000,00
2	iPhone 6S (Apple)	41000,00
3	iPhone 7 (Apple)	52000,00
4	Galaxy S8 (Samsung)	46000,00
5	Galaxy S8 Plus (Samsung)	56000,00
6	Mi6 (Xiaomi)	28000,00
7	OnePlus 5 (OnePlus)	38000,00

When executing this command, the table being sampled (in this case, ProductSummary) must not exist in the database.

But let's say we then decide to add all the data from the Products table to the already existing ProductSummary table. In this case, you can again use the INSERT command:

```

1 INSERT INTO ProductSummary
2 SELECT ProductName + ' (' + Manufacturer + ')' AS ModelName, Price
3 FROM Products

```

Here, the added values actually represent the result of a selection from the Products table.

## Sorting. ORDER BY

The ORDER BY operator allows you to sort the retrieved values by a specific column:

```

1 SELECT *
2 FROM Products
3 ORDER BY ProductName

```

In this case, the rows are sorted in ascending order by the value of the ProductName column:



SQLQuery3.sql - E...ENEPC\Eugene (52))\* X

```

1 USE productsdb;
2
3 SELECT *
4 FROM Products
5 ORDER BY ProductName

```

100 %

Results Messages

	Id	ProductName	Manufacturer	ProductCount	Price
1	4	Galaxy S8	Samsung	2	46000,00
2	5	Galaxy S8 Plus	Samsung	1	56000,00
3	1	iPhone 6	Apple	3	36000,00
4	2	iPhone 6S	Apple	2	41000,00
5	3	iPhone 7	Apple	5	52000,00
6	6	Mi6	Xiaomi	5	28000,00
7	7	OnePlus 5	OnePlus	6	38000,00

You can also sort by column alias, which is defined using the AS operator:

```

1 SELECT ProductName, ProductCount * Price AS TotalSum
2 FROM Products
3 ORDER BY TotalSum

```

SQLQuery3.sql - E...ENEPC\Eugene (52))\* X

```

1 USE productsdb;
2
3 SELECT ProductName, ProductCount * Price AS TotalSum
4 FROM Products
5 ORDER BY TotalSum

```

100 %

Results Messages

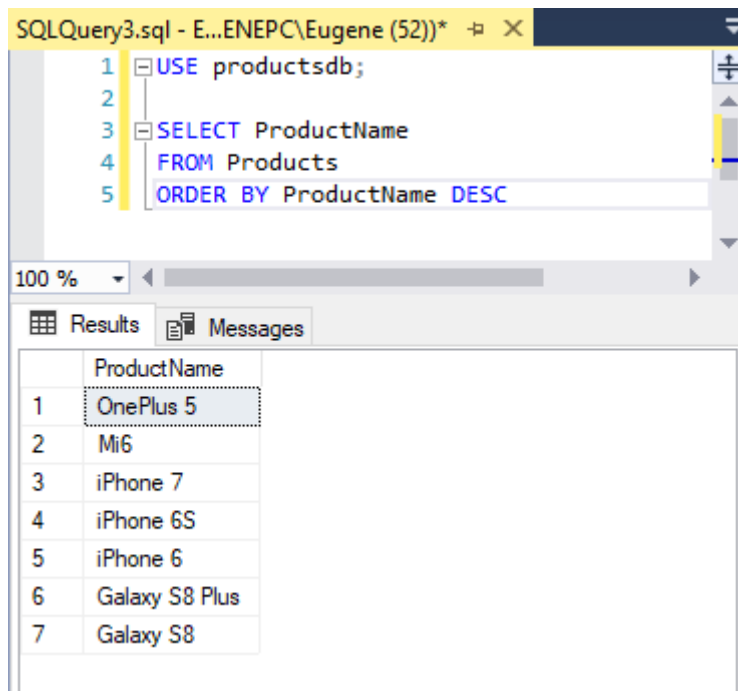
	ProductName	TotalSum
1	Galaxy S8 Plus	56000,00
2	iPhone 6S	82000,00
3	Galaxy S8	92000,00
4	iPhone 6	108000,00
5	Mi6	140000,00
6	OnePlus 5	228000,00
7	iPhone 7	260000,00

The default sort is ascending. You can use the optional DESC operator to sort in descending order.

```

1 SELECT ProductName
2 FROM Products
3 ORDER BY ProductName DESC

```



By default, the ASC operator is used instead of DESC:

```
1 SELECT ProductName  
2 FROM Products  
3 ORDER BY ProductName ASC
```

If you need to sort by several columns at once, then all of them are listed after the ORDER BY operator:

```
1 SELECT ProductName, Price, Manufacturer  
2 FROM Products  
3 ORDER BY Manufacturer, ProductName
```

In this case, the rows are first sorted by the Manufacturer column in ascending order. Then, if there are two rows in which the Manufacturer column has the same value, then they are sorted by the ProductName column, also in ascending order. But again, using ASC and DESC, you can separately define ascending and descending sorting for different columns:

```
1 SELECT ProductName, Price, Manufacturer  
2 FROM Products  
3 ORDER BY Manufacturer ASC, ProductName DESC
```

SQLQuery3.sql - E...ENEPC\Eugene (52))\*

```

1 USE productsdb;
2
3 SELECT ProductName, Price, Manufacturer
4 FROM Products
5 ORDER BY Manufacturer ASC, ProductName DESC

```

100 %

Results Messages

	ProductName	Price	Manufacturer
1	iPhone 7	52000,00	Apple
2	iPhone 6S	41000,00	Apple
3	iPhone 6	36000,00	Apple
4	OnePlus 5	38000,00	OnePlus
5	Galaxy S8 Plus	56000,00	Samsung
6	Galaxy S8	46000,00	Samsung
7	Mi6	28000,00	Xiaomi

You can also use a complex expression based on columns as a sort criterion:

```

1 SELECT ProductName, Price, ProductCount
2 FROM Products
3 ORDER BY ProductCount * Price

```

SQLQuery3.sql - E...ENEPC\Eugene (52))\*

```

1 USE productsdb;
2
3 SELECT ProductName, Price, ProductCount
4 FROM Products
5 ORDER BY ProductCount * Price

```

100 %

Results Messages

	ProductName	Price	ProductCount
1	Galaxy S8 Plus	56000,00	1
2	iPhone 6S	41000,00	2
3	Galaxy S8	46000,00	2
4	iPhone 6	36000,00	3
5	Mi6	28000,00	5
6	OnePlus 5	38000,00	6
7	iPhone 7	52000,00	5

## Extracting a Range of Rows

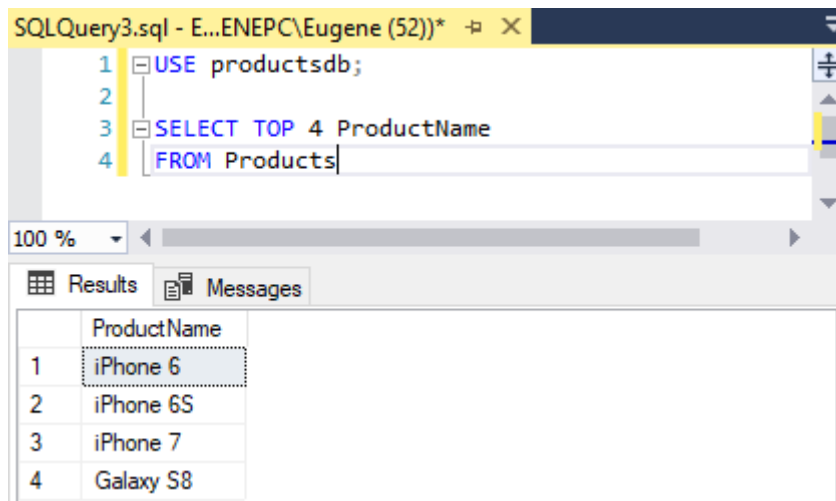
### TOP operator

The TOP operator allows you to select a certain number of rows from a table:

```

1 SELECT TOP 4 ProductName
2 FROM Products

```



The optional PERCENT operator allows you to select a percentage of rows from a table. For example, let's select 75% of the rows:

```
1 SELECT TOP 75 PERCENT Product Name  
2 FROM Products
```

## OFFSET and FETCH

The TOP operator allows you to extract a certain number of rows, starting from the beginning of the table. To retrieve a set of rows from anywhere, the OFFSET and FETCH operators are used. It is important that these operators only apply on the sorted dataset after the ORDER BY clause.

```
1 ORDER BY expression  
2 OFFSET offset_from_beginning {ROW|ROWS}  
3 [FETCH] {FIRST|NEXT} number_of_retrieval_rows {ROW|ROWS} ONLY]
```

For example, let's select all rows starting from the third one:

```
one SELECT * FROM Products  
2 ORDER BY ID  
3 OFFSET 2 ROWS;
```

The number after the OFFSET keyword specifies how many lines to skip.

SQLQuery3.sql - E...ENEPC\Eugene (52))\*

```

1 USE productsdb;
2
3 SELECT * FROM Products
4 ORDER BY Id
5 OFFSET 2 ROWS;

```

100 %

Results Messages

	Id	ProductName	Manufacturer	ProductCount	Price
1	3	iPhone 7	Apple	5	52000,00
2	4	Galaxy S8	Samsung	2	46000,00
3	5	Galaxy S8 Plus	Samsung	1	56000,00
4	6	Mi6	Xiaomi	5	28000,00
5	7	OnePlus 5	OnePlus	6	38000,00

Now select only three rows, starting with the third one:

```

1 SELECT * FROM Products
2 ORDER BY ID
3 OFFSET 2 ROWS
4 FETCH NEXT 3 ROWS ONLY;

```

The FETCH statement is followed by the FIRST or NEXT keyword (which doesn't matter in this case) and then the number of rows to be retrieved.

SQLQuery3.sql - E...ENEPC\Eugene (52))\*

```

1 USE productsdb;
2
3 SELECT * FROM Products
4 ORDER BY Id
5 OFFSET 2 ROWS
6 FETCH NEXT 3 ROWS ONLY;

```

100 %

Results Messages

	Id	ProductName	Manufacturer	ProductCount	Price
1	3	iPhone 7	Apple	5	52000,00
2	4	Galaxy S8	Samsung	2	46000,00
3	5	Galaxy S8 Plus	Samsung	1	56000,00

This combination of operators is usually used for page navigation when you need to get a specific page of data.

## Filtration. WHERE

The SELECT command uses the WHERE clause to filter. After this operator, a condition is set, which the string must meet:

```

1 WHERE condition

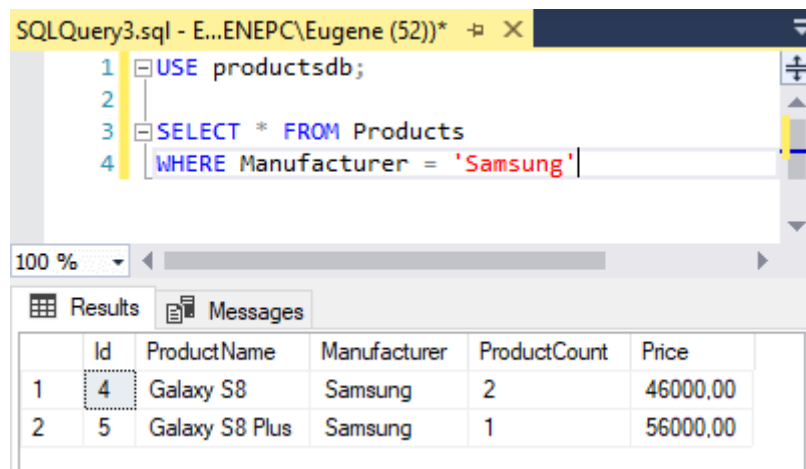
```

If the condition is true, then the string is included in the resulting selection. As you can use the comparison operations. These operations compare two expressions. The following comparison operations can be used in T-SQL:

- =: equality comparison (unlike C-like languages, T-SQL uses a single equals sign to compare for equality)
- <>: comparison for inequality
- <: less than
- >: more than
- !<: not less than
- !>: not more than
- <=: less than or equal
- >=: greater than or equal

For example, let's find all products manufactured by Samsung:

```
1  SELECT * FROM Products
2  WHERE Manufacturer = 'Samsung'
```



The screenshot shows a SQL query window titled 'SQLQuery3.sql - E...ENEPC\Eugene (52))' containing the following query:

```
1 USE productsdb;
2
3 SELECT * FROM Products
4 WHERE Manufacturer = 'Samsung'
```

Below the query window, the 'Results' tab is active, displaying a table with the following data:

	Id	ProductName	Manufacturer	ProductCount	Price
1	4	Galaxy S8	Samsung	2	46000,00
2	5	Galaxy S8 Plus	Samsung	1	56000,00

It is worth noting that in this case the case does not matter, and we could use the string "Samsung", and "SAMSUNG", and "samsung" for the search. All of these options would produce an equivalent sample result.

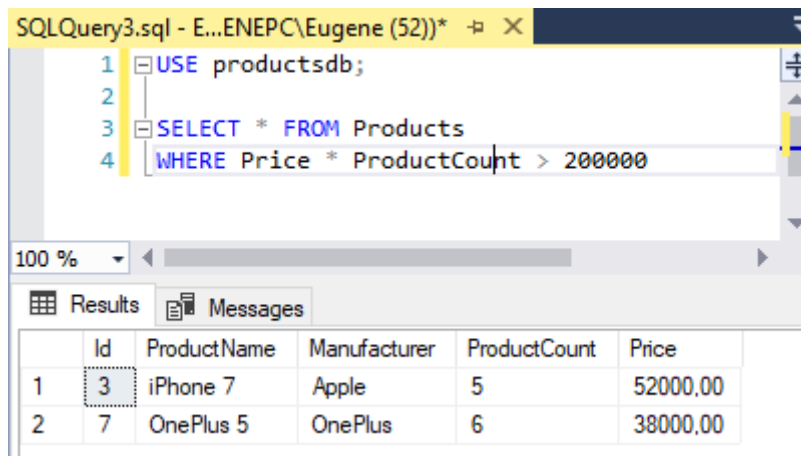
Another example is to find all products whose price is greater than 45000:

```
1  SELECT * FROM Products
2  WHERE Price > 45000
```

More complex expressions can also be used as a condition. For example, let's find all products whose total cost is more than 200,000:

```
1  SELECT * FROM Products
```

```
2 WHERE Price * ProductCount > 200000
```



## Logical operators

Logical operators can be used to combine multiple conditions into one. T-SQL has the following logical operators:

- **AND:** logical AND operation. It combines two expressions:  
one expression1 AND expression2
- Only if both of these expressions are true at the same time, then the general condition of the AND operator will also be true. That is, if both the first condition is true and the second.
- **OR:** logical OR operation. It also concatenates two expressions:  
one expression1 OR expression2
- If at least one of these expressions is true, then the general condition of the OR operator will also be true. That is, if either the first condition is true, or the second.
- **NOT:** logical negation operation. If the expression in this operation is false, then the general condition is true.  
one NOT expression

If these operators occur in the same expression, then NOT is performed first, then AND, and finally OR.

For example, let's select all products that have a manufacturer Samsung and at the same time the price is more than 50,000:

```
1 SELECT * FROM Products  
2 WHERE Manufacturer = 'Samsung' AND Price > 50000
```

SQLQuery3.sql - E...ENEPC\Eugene (52))\* X

```

1 USE productsdb;
2
3 SELECT * FROM Products
4 WHERE Manufacturer = 'Samsung' AND Price > 50000

```

100 %

Results Messages

	Id	ProductName	Manufacturer	ProductCount	Price
1	5	Galaxy S8 Plus	Samsung	1	56000,00

Now let's change the operator to OR. That is, we select all products that either have a Samsung manufacturer or a price greater than 50,000:

```

1 SELECT * FROM Products
2 WHERE Manufacturer = 'Samsung' OR Price > 50000

```

SQLQuery3.sql - E...ENEPC\Eugene (52))\* X

```

1 USE productsdb;
2
3 SELECT * FROM Products
4 WHERE Manufacturer = 'Samsung' OR Price > 50000

```

100 %

Results Messages

	Id	ProductName	Manufacturer	ProductCount	Price
1	3	iPhone 7	Apple	5	52000,00
2	4	Galaxy S8	Samsung	2	46000,00
3	5	Galaxy S8 Plus	Samsung	1	56000,00

Using the NOT operator - select all products whose manufacturer is not Samsung:

```

1 SELECT * FROM Products
2 WHERE NOT Manufacturer = 'Samsung'

```



SQLQuery3.sql - E...ENEPC\Eugene (52))\*

```

1 USE productsdb;
2
3 SELECT * FROM Products
4 WHERE NOT Manufacturer = 'Samsung'

```

100 %

Results Messages

	Id	ProductName	Manufacturer	ProductCount	Price
1	1	iPhone 6	Apple	3	36000,00
2	2	iPhone 6S	Apple	2	41000,00
3	3	iPhone 7	Apple	5	52000,00
4	6	Mi6	Xiaomi	5	28000,00
5	7	OnePlus 5	OnePlus	6	38000,00

But in most cases, it is quite possible to do without the NOT operator. So, in the previous example, we can rewrite it as follows:

```

1 SELECT * FROM Products
2 WHERE Manufacturer <> 'Samsung'

```

You can also use several statements in one SELECT command at once:

```

1 SELECT * FROM Products
2 WHERE Manufacturer = 'Samsung' OR Price > 30000 AND ProductCount > 2

```

Since the AND operator has a higher precedence, the Price > 30000 AND ProductCount > 2 subexpression will be executed first, and only then the OR operator. That is, goods are selected here that have more than 2 in stock and whose price is more than 30,000 at the same time, or those goods that are manufactured by Samsung.

SQLQuery3.sql - E...ENEPC\Eugene (52))\* SQLQuery2.sql - E...ENEPC\Eugene (51))\*

```

1 USE productsdb;
2
3 SELECT * FROM Products
4 WHERE Manufacturer = 'Samsung' OR Price > 30000 AND ProductCount > 2

```

100 %

Results Messages

	Id	ProductName	Manufacturer	ProductCount	Price
1	1	iPhone 6	Apple	3	36000,00
2	3	iPhone 7	Apple	5	52000,00
3	4	Galaxy S8	Samsung	2	46000,00
4	5	Galaxy S8 Plus	Samsung	1	56000,00
5	7	OnePlus 5	OnePlus	6	38000,00

With parentheses, we can also redefine the order of operations:

```
1  SELECT * FROM Products
2  WHERE (Manufacturer = 'Samsung' OR Price > 30000) AND ProductCount > 2
IS NULL
```

A number of columns may be nullable. This value is not equivalent to the empty string ". NULL represents the complete absence of any value. And to check for the presence of such a value, the IS NULL operator is used.

For example, let's select all products that do not have the ProductCount field set:

```
1  SELECT * FROM Products
2  WHERE ProductCount IS NULL
```

If, on the contrary, you need to get rows whose ProductCount field is not equal to NULL, then you can use the NOT operator:

```
1  SELECT * FROM Products
2  WHERE ProductCount IS NOT NULL
```

## Filter Operators

### IN operator

The IN operator allows you to define a set of values that the columns should have:

```
1  WHERE expression [NOT] IN (expression)
```

The parenthesized expression after IN defines a set of values. This set can be calculated dynamically based on, for example, another request, or it can be constant values.

For example, let's select products whose manufacturer is either Samsung, or Xiaomi, or Huawei:

```
1  SELECT * FROM Products
2  WHERE Manufacturer IN ('Samsung', 'Xiaomi', 'Huawei')
```

The screenshot shows a SQL query window titled 'SQLQuery3.sql - E...ENEPC\Eugene (52))'. The query is as follows:

```

1 USE productsdb;
2
3 SELECT * FROM Products
4 WHERE Manufacturer IN ('Samsung', 'Xiaomi', 'Huawei')

```

Below the query window, the 'Results' tab is active, displaying a table with the following data:

	Id	ProductName	Manufacturer	ProductCount	Price
1	4	Galaxy S8	Samsung	2	46000,00
2	5	Galaxy S8 Plus	Samsung	1	56000,00
3	6	Mi6	Xiaomi	5	28000,00

We could also check all these values through the OR operator:

```

1 SELECT * FROM Products
2 WHERE Manufacturer = 'Samsung' OR Manufacturer = 'Xiaomi' OR Manufacturer =
  'Huawei'

```

But using the IN operator is much more convenient, especially if there are a lot of similar values.

Using the NOT operator, you can find all rows that, on the contrary, do not match a set of values:

```

1 SELECT * FROM Products
2 WHERE Manufacturer NOT IN ('Samsung', 'Xiaomi', 'Huawei')

```

## BETWEEN Operator

The BETWEEN operator defines a range of values with a start and end value that the expression must match:

```

1 WHERE statement [NOT] BETWEEN start_value AND end_value

```

For example, let's get all products with a price between 20,000 and 40,000 (the start and end values are also included in the range):

```

1 SELECT * FROM Products
2 WHERE Price BETWEEN 20000 AND 40000

```

SQLQuery3.sql - E...ENEPC\Eugene (52))

```

1 USE productsdb;
2
3 SELECT * FROM Products
4 WHERE Price BETWEEN 20000 AND 40000

```

100 %

Results Messages

	Id	ProductName	Manufacturer	ProductCount	Price
1	1	iPhone 6	Apple	3	36000,00
2	6	Mi6	Xiaomi	5	28000,00
3	7	OnePlus 5	OnePlus	6	38000,00

If, on the contrary, it is necessary to select those rows that do not fall into this range, then the NOT operator is used:

```

1 SELECT * FROM Products
2 WHERE Price NOT BETWEEN 20000 AND 40000

```

You can also use more complex expressions. For example, let's get goods, the stocks of which are for a certain amount (price \* quantity):

```

1 SELECT * FROM Products
2 WHERE Price * ProductCount BETWEEN 100000 AND 200000

```

## LIKE operator

The LIKE operator accepts a string pattern that the expression must match.

```

1 WHERE expression [NOT] LIKE string_pattern

```

A number of special wildcard characters can be used to define a pattern:

- %: matches any substring, which can have any number of characters, while the substring may not contain a single character
- \_: matches any single character
- [ ]: matches a single character, which is given in square brackets
- [-]: matches one character from a specific range
- [^]: matches a single character that is not specified after the ^ character

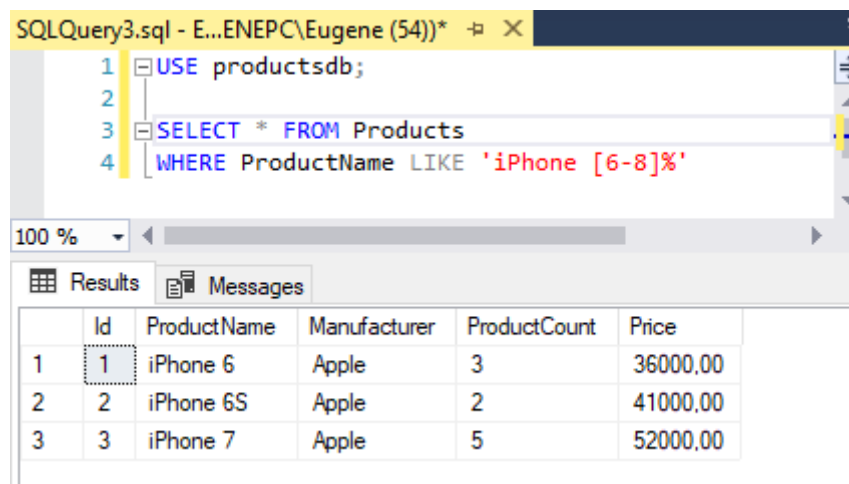
Some examples of using substitutions:

- WHERE ProductName LIKE 'Galaxy%'  
Corresponds to values such as "Galaxy Ace 2" or "Galaxy S7"
- WHERE ProductName LIKE 'Galaxy S\_'  
Corresponds to values such as "Galaxy S7" or "Galaxy S8"

- WHERE ProductName LIKE 'iPhone[78]'  
Matches values such as "iPhone 7" or "iPhone8"
- WHERE ProductName LIKE 'iPhone [6-8]'  
Matches values such as "iPhone 6", "iPhone 7", or "iPhone8"
- WHERE ProductName LIKE 'iPhone [^7]%'  
Matches values such as "iPhone 6", "iPhone 6S", or "iPhone8". But does not match "iPhone 7" and "iPhone 7S"
- WHERE ProductName LIKE 'iPhone [^1-6]%'  
Corresponds to values such as "iPhone 7", "iPhone 7S" and "iPhone 8". But does not match "iPhone 5", "iPhone 6" and "iPhone 6S"

Let's use the LIKE operator:

```
1  SELECT * FROM Products
2  WHERE ProductName LIKE 'iPhone[6-8]%'
```



SQLQuery3.sql - E:\ENEPC\Eugene (54)\*

```
1  USE productsdb;
2
3  SELECT * FROM Products
4  WHERE ProductName LIKE 'iPhone [6-8]%'
```

100 %

Results Messages

	Id	ProductName	Manufacturer	ProductCount	Price
1	1	iPhone 6	Apple	3	36000,00
2	2	iPhone 6S	Apple	2	41000,00
3	3	iPhone 7	Apple	5	52000,00

## Data update. UPDATE command

The UPDATE command is used to update existing rows in a table. It has the following formal syntax:

```
1  UPDATE table_name
2  SET column1 = value1, column2 = value2, ... columnN = valueN
3  [FROM selection AS selection_alias]
four[WHERE update_condition]
```

For example, let's increase the price of all products by 5000:

```
1  UPDATE Products
2  SET Price = Price + 5000
```

SQLQuery3.sql - E...ENEPC\Eugene (52))\*

```

1 USE productsdb;
2
3 SELECT * FROM Products
4
5 UPDATE Products
6 SET Price = Price + 5000
7
8 SELECT * FROM Products

```

100 %

Results Messages

	Id	ProductName	Manufacturer	ProductCount	Price
1	1	iPhone 6	Apple	3	36000,00
2	2	iPhone 6S	Apple	2	41000,00
3	3	iPhone 7	Apple	5	52000,00
4	4	Galaxy S8	Samsung	2	46000,00
5	5	Galaxy S8 Plus	Samsung	1	56000,00
6	6	Mi6	Xiaomi	5	28000,00
7	7	OnePlus 5	OnePlus	6	38000,00

---

	Id	ProductName	Manufacturer	ProductCount	Price
1	1	iPhone 6	Apple	3	41000,00
2	2	iPhone 6S	Apple	2	46000,00
3	3	iPhone 7	Apple	5	57000,00
4	4	Galaxy S8	Samsung	2	51000,00
5	5	Galaxy S8 Plus	Samsung	1	61000,00
6	6	Mi6	Xiaomi	5	33000,00
7	7	OnePlus 5	OnePlus	6	43000,00

Use the criteria, and change the manufacturer name from "Samsung" to "Samsung Inc.":

```

1 UPDATE Products
2 SET Manufacturer = 'Samsung Inc.'
3 WHERE Manufacturer = 'Samsung'

```

A more complex query - replace the Manufacturer field with the value "Apple" with "Apple Inc." in the first 2 lines:

```

1 UPDATE Products
2 SET Manufacturer = 'Apple Inc.'
3 FROM
4 (SELECT TOP 2 FROM Products WHERE Manufacturer='Apple') AS Selected
5 WHERE Products.Id = Selected.Id

```

A subquery after the FROM keyword fetches the first two rows where Manufacturer='Apple'. The alias Selected will be defined for this selection. The alias is specified after the AS statement.

Next comes the update condition `Products.Id = Selected.Id`. That is, in fact, we are dealing with two tables - `Products` and `Selected` (which is derived from `Products`). `Selected` contains the first two lines where `Manufacturer='Apple'`. In `Products` - generally all lines. And the update is performed only for those rows that are in the `Selected` selection. That is, if there are dozens of products with Apple manufacturer in the `Products` table, then the update will affect only the first two of them.

## Deleting data. DELETE command

To delete, use the `DELETE` command:

```
1  DELETE [FROM] table_name
2  WHERE delete_condition
```

For example, let's delete the rows whose id is 9:

```
1  DELETE Products
2  WHERE Id=9
```

Or delete all products manufactured by Xiaomi and priced less than 15000:

```
1  DELETE Products
2  WHERE Manufacturer='Xiaomi' AND Price < 15000
```

A more complex example - let's remove the first two products whose manufacturer is Apple:

```
1  DELETE Products FROM
2  (SELECT TOP 2 * FROM Products
3   WHERE Manufacturer='Apple') AS Selected
4  WHERE Products.Id = Selected.Id
```

After the first `FROM` statement, two rows are fetched from the `Products` table. This selection is assigned the alias `Selected` using the `AS` statement. Next, we set the condition that if the `Id` in the `Products` table has the same value as the `Id` in the `Selected` selection, then the row is deleted.

SQLQuery1.sql - E...ENEPC\Eugene (53))\* X

```

1 USE productsdb;
2
3 SELECT * FROM Products
4
5 DELETE Products FROM
6 (SELECT TOP 2 * FROM Products
7  WHERE Manufacturer='Apple') AS Selected
8  WHERE Products.Id = Selected.Id
9
10 SELECT * FROM Products
11

```

100 %

Results Messages

	Id	ProductName	Manufacturer	ProductCount	Price
1	1	iPhone 6	Apple	3	36000,00
2	2	iPhone 6S	Apple	2	41000,00
3	3	iPhone 7	Apple	5	52000,00
4	4	Galaxy S8	Samsung	2	46000,00
5	5	Galaxy S8 Plus	Samsung	1	56000,00
6	6	Mi6	Xiaomi	5	28000,00
7	7	OnePlus 5	OnePlus	6	38000,00

	Id	ProductName	Manufacturer	ProductCount	Price
1	3	iPhone 7	Apple	5	52000,00
2	4	Galaxy S8	Samsung	2	46000,00
3	5	Galaxy S8 Plus	Samsung	1	56000,00
4	6	Mi6	Xiaomi	5	28000,00
5	7	OnePlus 5	OnePlus	6	38000,00

If it is necessary to completely delete all rows, regardless of the condition, then the condition can be omitted:

```
1 DELETE Products
```