Laboratory work

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

Purpose: Create and delete a database. Creating and deleting tables. T-SQL data types. Attributes and restrictions of columns and tables. foreign keys. Change table. Packages. GO command.

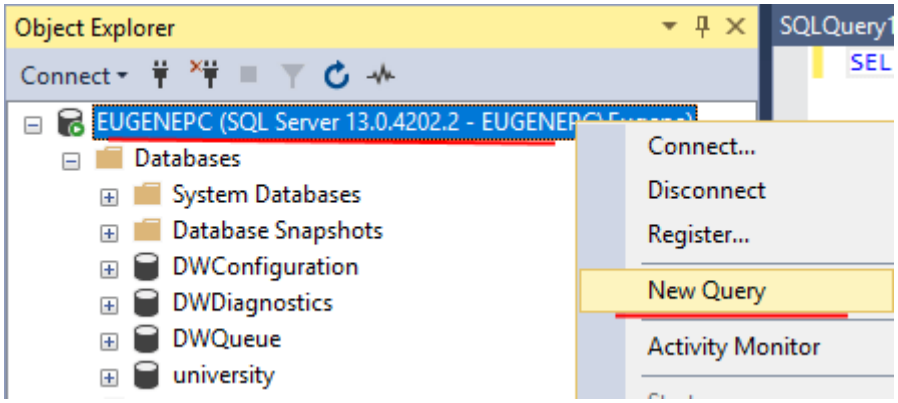
Progress

Creating and deleting a database

Database creation

The CREATE DATABASE command is used to create a database.

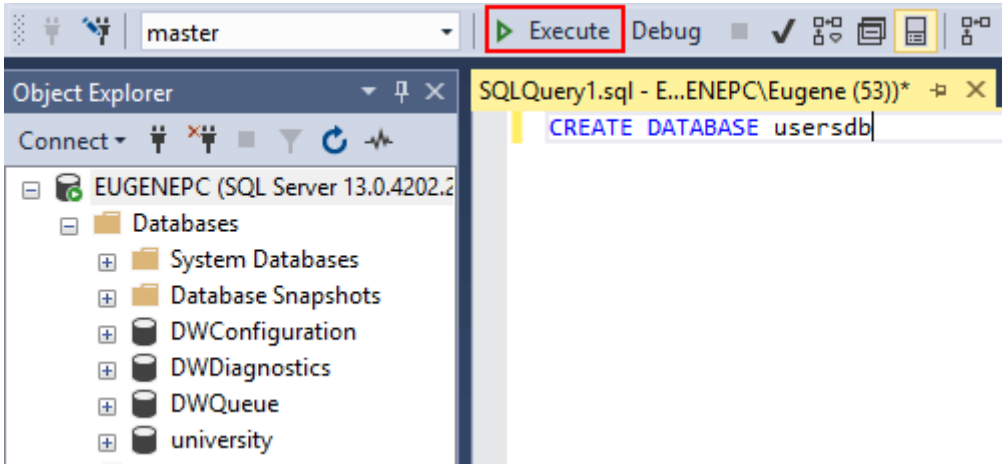
To create a new database, open SQL Server Management Studio. Click on the server assignment in the Object Explorer window and select New Query from the menu that appears.



In the central field for entering sql expressions, enter the following code:

```
1   CREATE DATABASE usersdb
```

This way we create a database that will be called "usersdb":



To execute the command, click the Execute button on the toolbar or the F5 key. And a new database will appear on the server.

After creating the database, we can set it as the current one with the USE command:

```
1 USE usersdb;
```

Attaching a database

It is possible that we already have a database file, which, for example, was created on another computer. The database file is a file with the mdf extension, and in principle we can transfer this file. However, even if we copy his computer with MS SQL Server installed, the copied database will not appear on the server just like that. To do this, you must attach the database to the server. In this case, the expression is applied:

```
1 CREATE DATABASE db_name
2 ON PRIMARY(FILENAME=file_path')
3 FOR ATTACH;
```

It is better to use the directory where the rest of the server's databases are stored as the directory for the database. For example, let's say in our case the data file is called userstoredb.mdf. And we want to add this file to the server as a database. First, it must be copied to the above directory. Then, to attach the database to the server, use the following command:

```
1   CREATE DATABASE contactsdb
2   ON PRIMARY(FILENAME='your_file_path')
3   FOR ATTACH;
```

After executing the command, the contactsdb database will appear on the server.

Deleting a database

To drop a database, use the DROP DATABASE command, which has the following syntax:

```
1   DROP DATABASE database_name1[,database_name2]...
```

After the command, separated by commas, we can list all databases to be deleted. For example, deleting the contactsdb database:

```
1  DROP DATABASE contactsdb
```

It should be taken into account that even if the database being deleted was attached, all database files will still be deleted.

Creating and deleting tables

The CREATE TABLE command is used to create tables. You can use a number of statements with this command that define table columns and their attributes. And besides, you can use a number of operators that define the properties of the table as a whole. One database can contain up to 2 billion tables.

The general syntax for creating a table is as follows:

```
1   CREATE TABLE table_name
2   (column_name1 data_type column_attributes1,
3   column_name2 data_type column_attributes2,
4   ..........................................
5   column_nameN data_type column_attributesN,
6   table_attributes
7   )
```

The CREATE TABLE command is followed by the name of the table to be created. The table name acts as its identifier in the database, so it must be unique. The name must be no longer than 128 characters. The name can consist of alphanumeric characters, as well as $ and underscores. The first character must be a letter or an underscore.

The object name cannot include spaces and cannot represent one of the Transact-SQL keywords. If the identifier does contain whitespace characters, then it must be enclosed in quotation marks. If it is necessary to use keywords as a name, then these words are placed in square brackets.

Examples of valid identifiers:

```
1   Users
2   tags$345
3   users_accounts
4   "user accounts"
5   [Table]
```

After the table name, in brackets, the parameters of all columns are indicated and at the very end, the attributes that apply to the entire table. Column attributes and table attributes are optional and can be omitted.

In its simplest form, the CREATE TABLE command must contain at least the table name, column names, and column types.

A table can contain from 1 to 1024 columns. Each column must have a unique name within the current table and must be assigned a data type.

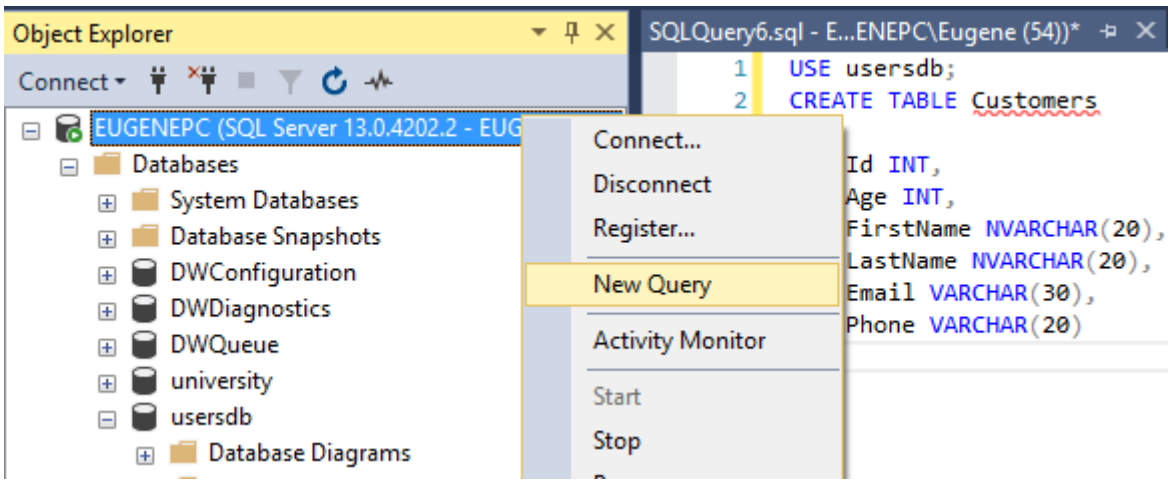
For example, the definition of the simplest table Customers:

```
1 CREATE TABLE Customers
2 (
3 ID INT,
4 Age INT,
5 FirstName NVARCHAR(20),
6 LastName NVARCHAR(20),
7 Email VARCHAR(30),
8 Phone VARCHAR(20)
9 )
```

In this case, six columns are defined in the Customers table: Id, FirstName, LastName, Age, Email, Phone. The first two columns represent the client ID and age and are of type INT, meaning they will store numeric values. The next two columns represent the customer's first and last name and are of type NVARCHAR(20), which means they represent a UNICODE string with a maximum of 20 characters. The last two columns Email and Phone represent the customer's email address and phone number and are of type VARCHAR(30/20) - they also store a string, but not in UNICODE encoding.

Creating a table in SQL Management Studio

Let's create a simple table on the server. To do this, open SQL Server Management Studio and right-click on the server name. In the context menu that appears, select the New Query item.

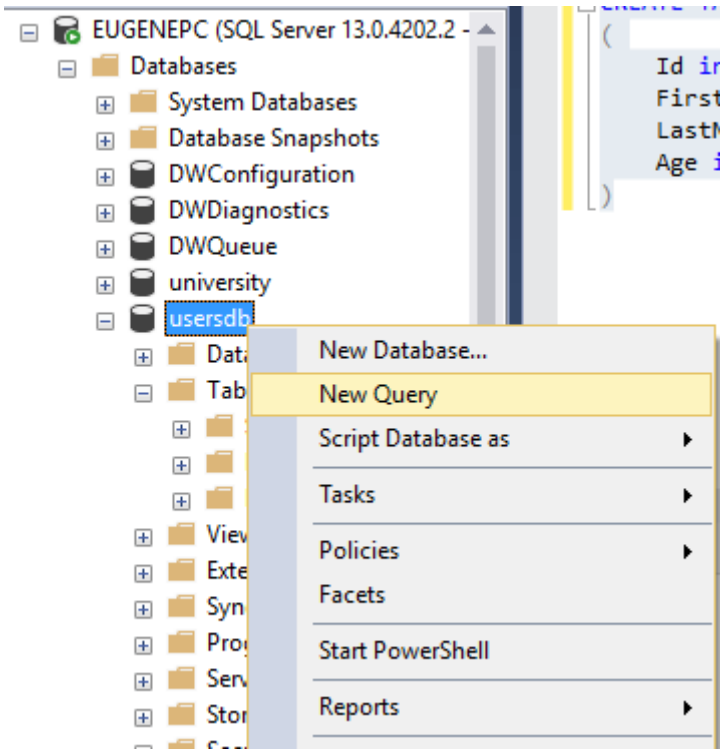


The table is created within the current database. If we launch the SQL editor window as it was done above - from under the name of the server, then the default database is not installed. And to install it, you must use the USE command, after which the name of the database is indicated. Therefore, we will enter the following expressions in the field of the SQL command editor:

```
USE usersdb;

CREATE TABLE Customers
(
ID INT,
Age INT,
FirstName NVARCHAR(20),
LastName NVARCHAR(20),
Email VARCHAR(30),
Phone VARCHAR(20)
);
```

That is, the Customers table, which was discussed earlier, is added to the database. You can also open the editor from under the database by also right-clicking on it and selecting New Query:



In this case, the database under which the editor was opened will be considered as the current one, and it will not be required to additionally install it using the USE command.

Deleting tables

To drop tables, use the DROP TABLE command, which has the following syntax:

```
1 DROP TABLE table1[, table2, ...]
```

For example, deleting the Customers table:

```
1 DROP TABLE Customers
```

Renaming a table

The "sp_rename" system stored procedure is used to rename tables. For example, renaming the Users table to UserAccounts in the usersdb database:

```
1 USE usersdb;
2 EXEC sp_rename 'Users', 'UserAccounts';
```

T-SQL Data Types

When you create a table, you must specify a specific data type for all of its columns. The data type determines what values can be stored in the column, how much they will take up memory space.

The T-SQL language provides many different types. Depending on the nature of the values, all of them can be divided into groups.

Numeric Data Types

- **bit**: stores a value from 0 to 16. Can act as an analogue of a boolean type in programming languages (in this case, the value of true corresponds to 1, and the value of false - 0). For values up to 8 (inclusive) it takes 1 byte, for values from 9 to 16 - 2 bytes.
- **TINYINT**: stores numbers from 0 to 255. Occupies 1 byte. Good for storing small numbers.
- **SMALLINT**: stores numbers from -32768 to 32767. Occupies 2 bytes
- **INT**: stores numbers from -2147483648 to 2147483647. Occupies 4 bytes. The most used type for storing numbers.
- **BIGINT**: stores very large numbers from -9 223 372 036 854 775 808 to 9 223 372 036 854 775 807, which occupy 8 bytes of memory.
- **DECIMAL**: stores fixed-precision numbers. It takes from 5 to 17 bytes, depending on the number of numbers after the decimal point.

  This type can take two parameters precision and scale: DECIMAL(precision, scale). The precision parameter represents the maximum number of digits that the number can store. This value must be between 1 and 38. The default is 18.

  The scale parameter represents the maximum number of digits that a number after the decimal point can contain. This value must be between 0 and the value of the precision parameter. It defaults to 0.
- **NUMERIC**: This type is similar to the DECIMAL type.
- **SMALLMONEY**: stores fractional values from -214 748.3648 to 214 748.3647. Designed to store money. Takes 4 bytes. Equivalent to DECIMAL(10,4).
- **MONEY**: Stores fractional values from -922337203685477.5808 to 922337203685477.5807. Represents monetary values and occupies 8 bytes. Equivalent to DECIMAL(19,4).
- **FLOAT**: stores numbers from -1.79E+308 to 1.79E+308. It takes from 4 to 8 bytes depending on the fractional part.

May be defined as FLOAT(n), where n represents the number of bits that are used to store the decimal part of the number (the mantissa). By default n = 53.

- **REAL**: stores numbers from -340E+38 to 3.40E+38. Takes 4 bytes. Equivalent to the FLOAT(24) type.

Examples of numeric columns:

```
1   salary money,
2   TotalWeight DECIMAL(9,2),
3   Age INT,
4   Surplus FLOAT
```

Data Types Representing Date and Time

- **DATE**: stores dates from 0001-01-01 (January 1, 0001) to 9999-12-31 (December 31, 9999). Takes 3 bytes.
- **TIME**: stores the time in the range from 00:00:00.0000000 to 23:59:59.9999999. It takes from 3 to 5 bytes.

  May be of the form TIME(n), where n represents the number of digits from 0 to 7 in fractional seconds.
- **DATETIME**: stores dates and times from 01/01/1753 to 12/31/9999. Occupies 8 bytes.
- **DATETIME2**: Stores dates and times in the range 01/01/0001 00:00:00.0000000 to 12/31/9999 23:59:59.9999999. It takes from 6 to 8 bytes depending on the time precision.

  May be of the form DATETIME2(n), where n represents the number of digits from 0 to 7 in fractional seconds.
- **SMALLDATETIME**: Stores dates and times between 01/01/1900 and 06/06/2079, which are the nearest dates. Takes 4 bytes.
- **DATETIMEOFFSET**: Stores dates and times in the range 0001-01-01 to 9999-12-31. Stores detailed time information accurate to 100 nanoseconds. Takes 10 bytes.

Common date formats:

- yyyy-mm-dd- 2017-07-12
- dd/mm/yyyy- 12/07/2017
- mm-dd-yy- 07-12-17

In this format, two-digit numbers from 00 to 49 are treated as dates in the range 2000-2049. And the numbers from 50 to 99 as a range of numbers 1950 - 1999.

- Month dd, yyyyJuly 12, 2017

  Common time formats:
- hh:mi- 13:21
- hh:mi am/pm- 1:21 p.m.
- hh:mi:ss- 1:21:34
- hh:mi:ss:mmm- 1:21:34:12
- hh:mi:ss:nnnnnnn- 1:21:34:1234567

  String data types
- **CHAR**: stores a string between 1 and 8,000 characters long. Allocates 1 byte for each character. Not suitable for many languages because it stores non-Unicode characters.

  The number of characters a column can store is passed in parentheses. For example, for a CHAR(10) column, 10 bytes will be allocated. And if we store a string of less than 10 characters in a column, then it will be padded with spaces.
- **VARCHAR**: stores a string. 1 byte is allocated for each character. You can specify a specific length for a column, from 1 to 8,000 characters, such as VARCHAR(10). If the string must have more than 8000 characters, then the MAX size is set, and up to 2 GB can be allocated for storing the string: VARCHAR(MAX).

  Not suitable for many languages because it stores non-Unicode characters.

  Unlike the CHAR type, if a 5-character string is stored in a VARCHAR(10) column, exactly five characters will be stored in the column.
- **NCHAR**: Stores a Unicode string between 1 and 4,000 characters long. 2 bytes are allocated for each character. For example, NCHAR(15)
- **NVARCHAR**: stores a Unicode string. 2 bytes are allocated for each character. You can set a specific size from 1 to 4,000 characters: . If the string must have more than 4000 characters, then the MAX size is set, and up to 2 GB can be allocated for storing the string.

The other two types, TEXT and NTEXT, are deprecated and therefore not recommended. Instead, VARCHAR and NVARCHAR are used, respectively.

Examples of defining string columns:

```
1   Email VARCHAR(30),
2   Comment NVARCHAR(MAX)
```

Binary data types

- **BINARY**: stores binary data as a sequence of 1 to 8,000 bytes.
- **VARBINARY**: Stores binary data as a sequence of 1 to 8,000 bytes, or up to 2^31-1 bytes when using MAX (VARBINARY(MAX)).

  Another binary type, the IMAGE type, is deprecated and it is recommended to use the VARBINARY type instead.

  Other data types
- **UNIQUEIDENTIFIER**: a unique GUID (essentially a string with a unique value) that takes up 16 bytes.
- **TIMESTAMP**: some number that stores the version number of the row in the table. Occupies 8 bytes.
- **CURSOR**: represents a set of rows.
- **HIERARCHYID**: represents a position in the hierarchy.
- **SQL_VARIANT**: can store data of any other T-SQL data type.
- **XML**: Stores XML documents or fragments of XML documents. Takes up to 2 GB of memory.
- **TABLE**: represents the table definition.
- **GEOGRAPHY**: stores geographic data such as latitude and longitude.
- **GEOMETRY**: stores the coordinates of the location on the plane.

Column and table attributes and restrictions

When creating columns in T-SQL, we can use a number of attributes, a number of which are constraints. Let's take a look at these attributes.

PRIMARY KEY

The PRIMARY KEY expression can make a column the primary key.

```
1   CREATE TABLE Customers
```

```
2   (
3   ID INT PRIMARY key,
4   Age INT,
5   FirstName NVARCHAR(20),
6   LastName NVARCHAR(20),
7   Email VARCHAR(30),
8   Phone VARCHAR(20)
9   )
```

The primary key uniquely identifies a row in a table. Columns of type int do not have to be primary keys, they can represent any other type.

Setting the primary key at the table level:

```
1    CREATE TABLE Customers
2    (
3    ID INT,
4    Age INT,
5    FirstName NVARCHAR(20),
6    LastName NVARCHAR(20),
7    Email VARCHAR(30),
8    Phone VARCHAR(20),
9    PRIMARY KEY(Id)
10   )
```

The primary key can be a compound key. Such a key may be required if we have two columns at once that must uniquely identify a row in the table. For example:

```
1   CREATE TABLE OrderLines
2   (
3   OrderId INT,
4   ProductId INT,
5   Quantity INT,
6   price money,
7   PRIMARY KEY(OrderId, ProductId)
8   )
```

Here the fields OrderId and ProductId together act as a composite primary key. That is, there cannot be two rows in the OrderLines table where both of these fields would have the same values at the same time.

IDENTITY

The IDENTITY attribute allows you to make a column an identifier. This attribute can be assigned to INT, SMALLINT, BIGINT, TYNIINT, DECIMAL, and NUMERIC columns. When adding new data to a table, SQL Server will increment

the value of this column by one for the last record. Typically, the same column as the primary key acts as the identifier, although this is not required in principle.

```
1   CREATE TABLE Customers
2   (
3   ID INT PRIMARY key IDENTITY,
4   Age INT,
5   FirstName NVARCHAR(20),
6   LastName NVARCHAR(20),
7   Email VARCHAR(30),
8   Phone VARCHAR(20)
9   )
```

You can also use the full form of the attribute:

one IDENTITY(seed, increment)

Here, the seed parameter indicates the initial value from which the countdown will begin. And the increment parameter determines how much the next value will increase. By default, the attribute uses the following values:

```
one IDENTITY(1, 1)
```

That is, the countdown starts from 1. And subsequent values increase by one. But we can override this behavior. For example:

one ID INT IDENTITY(2, 3)

In this case, the countdown will start from 2, and the value of each subsequent entry will increase by 3. That is, the first line will have a value of 2, the second - 5, the third - 8, and so on.

Also note that only one column in a table should have this attribute.

UNIQUE

If we want a column to have only unique values, then we can define the UNIQUE attribute for it.

```
1   CREATE TABLE Customers
2   (
3   ID INT PRIMARY key IDENTITY,
4   Age INT,
5   FirstName NVARCHAR(20),
6   LastName NVARCHAR(20),
7   Email VARCHAR(30) UNIQUE,
8   Phone VARCHAR(20) UNIQUE
```

```
9   )
```

In this case, the columns that represent email address and phone number will have unique values. And we will not be able to add two rows to the table that have the same values for these columns.

We can also define this attribute at the table level:

```
1    CREATE TABLE Customers
2    (
3    ID INT PRIMARY key IDENTITY,
4    Age INT,
5    FirstName NVARCHAR(20),
6    LastName NVARCHAR(20),
7    Email VARCHAR(30),
8    Phone VARCHAR(20),
9    UNIQUE(Email, Phone)
10   )
```

NULL and NOT NULL

To specify whether a column can be null, you can set the column's NULL or NOT NULL attribute when you define the column. If this attribute is not explicitly used, the column will default to null. The exception is when the column is acting as a primary key, in which case the column defaults to NOT NULL.

```
1   CREATE TABLE Customers
2   (
3   ID INT PRIMARY key IDENTITY,
4   Age INT,
5   FirstName NVARCHAR(20) NOT NULL,
6   LastName NVARCHAR(20) NOT NULL,
7   Email VARCHAR(30) UNIQUE,
8   Phone VARCHAR(20) UNIQUE
9   )
```

DEFAULT

The DEFAULT attribute specifies the default value for the column. If no value is provided for a column when data is added, the default value will be used for that column.

```
1   CREATE TABLE Customers
2   (
3   ID INT PRIMARY key IDENTITY,
4   Age INT DEFAULT eighteen,
```

```
5   FirstName NVARCHAR(20) NOT NULL,
6   LastName NVARCHAR(20) NOT NULL,
7   Email VARCHAR(30) UNIQUE,
8   Phone VARCHAR(20) UNIQUE
9   );
```

Here, the Age column defaults to 18.

CHECK

The CHECK keyword specifies a limit on the range of values that can be stored in a column. To do this, after the word CHECK, a condition is indicated in brackets that a column or several columns must correspond to. For example, the age of clients cannot be less than 0 or greater than 100:

```
1   CREATE TABLE Customers
2   (
3   ID INT PRIMARY key IDENTITY,
4   Age INT DEFAULT 18 CHECK(Age >0 AND Age < 100),
5   FirstName NVARCHAR(20) NOT NULL,
6   LastName NVARCHAR(20) NOT NULL,
7   Email VARCHAR(30) UNIQUE CHECK(Email !=''),
8   Phone VARCHAR(20) UNIQUE CHECK(Phone !='')
9   );
```

It also specifies that the Email and Phone columns cannot have an empty string as a value (an empty string is not equivalent to NULL).

The AND keyword is used to connect conditions. Conditions can be specified as greater than (>), less than (<), not equal (!=) comparison operations.

You can also use CHECK to create a constraint on the table as a whole:

```
1    CREATE TABLE Customers
2    (
3    ID INT PRIMARY key IDENTITY,
4    Age INT DEFAULT eighteen,
5    FirstName NVARCHAR(20) NOT NULL,
6    LastName NVARCHAR(20) NOT NULL,
7    Email VARCHAR(30) UNIQUE,
8    Phone VARCHAR(20) UNIQUE,
9    CHECK((Age >0 AND Age<100) AND (Email !='') AND (Phone !=''))
10   )
```

CONSTRAINT statement. Setting the constraint name.

The CONSTRAINT keyword can be used to give a name to the constraints. PRIMARY KEY, UNIQUE, DEFAULT, CHECK can be used as constraints.

Constraint names can be set at the column level. They are specified after CONSTRAINT before attributes:

```
1    CREATE TABLE Customers
2    (
3    ID INT CONSTRAINT PK_Customer_Id PRIMARY key IDENTITY,
4    Age INT
5    CONSTRAINT DF_Customer_Age DEFAULT eighteen
6    CONSTRAINT CK_Customer_Age CHECK(Age >0 AND Age < 100),
7    FirstName NVARCHAR(20) NOT NULL,
8    LastName NVARCHAR(20) NOT NULL,
9    Email VARCHAR(30) CONSTRAINT UQ_Customer_Email UNIQUE,
10   Phone VARCHAR(20) CONSTRAINT UQ_Customer_Phone UNIQUE
11   )
```

Restrictions can have arbitrary names, but the following prefixes are generally used:

- "PK_" - for PRIMARY KEY
- "FK_" - for FOREIGN KEY
- "CK_" - for CHECK
- "UQ_" - for UNIQUE
- "DF_" - for DEFAULT

In principle, it is not necessary to specify constraint names; when setting the appropriate attributes, SQL Server automatically determines their names. But, knowing the name of the constraint, we can refer to it, for example, to remove it.

And you can also set all the names of the constraints through table attributes:

```
1    CREATE TABLE Customers
2    (
3    ID INT IDENTITY,
4    Age INT CONSTRAINT DF_Customer_Age DEFAULT eighteen,
5    FirstName NVARCHAR(20) NOT NULL,
6    LastName NVARCHAR(20) NOT NULL,
7    Email VARCHAR(30),
8    Phone VARCHAR(20),
9    CONSTRAINT PK_Customer_Id PRIMARY key (Id)
10   CONSTRAINT CK_Customer_Age CHECK(Age >0 AND Age < 100),
```

```
11   CONSTRAINT UQ_Customer_Email UNIQUE (Email)
12   CONSTRAINT UQ_Customer_Phone UNIQUE (Phone)
13   )
```

Foreign keys

Foreign keys are used to establish relationships between tables. A foreign key is set on columns from a dependent, subordinate table, and points to one of the columns from the master table. Although, as a rule, a foreign key points to a primary key from the associated master table, this does not have to be a sine qua non. The foreign key can also point to some other column that has a unique value.

The general syntax for setting a foreign key at the column level is:

```
1   [FOREIGN] KEY] REFERENCES main_table (column_of_main_table)
2   [ON DELETE {CASCADE|NO ACTION}]
3   [ON UPDATE {CASCADE|NO ACTION}]
```

To create a column-level foreign key constraint, the REFERENCES keyword is followed by the name of the related table and, in parentheses, the name of the related column that the foreign key will point to. Also, FOREIGN KEY keywords are usually added, but in principle they are not required to be specified. After the REFERENCES clause comes the ON DELETE and ON UPDATE clause.

The general syntax for setting a foreign key at the table level is:

```
1 FOREIGN key (column1,column2, ...columnN)
2 REFERENCES   main_table   (main_table_column1,   main_table_column2,   ...
  main_table_columnN)
3 [ON DELETE {CASCADE|NO ACTION}]
4 [ON UPDATE {CASCADE|NO ACTION}]
```

For example, let's define two tables and link them with a foreign key:

```
1    CREATE TABLE Customers
2    (
3    ID INT PRIMARY key IDENTITY,
4    Age INT DEFAULT eighteen,
5    FirstName NVARCHAR(20) NOT NULL,
6    LastName NVARCHAR(20) NOT NULL,
7    Email VARCHAR(30) UNIQUE,
8    Phone VARCHAR(20) UNIQUE
9    );
10
```

```
11   CREATE TABLE orders
12   (
13   ID INT PRIMARY key IDENTITY,
14   CustomerId INT REFERENCES Customers (Id),
15   CreatedAtDate
16   );
```

The tables Customers and Orders are defined here. Customers is the main one and represents the customer. Orders is dependent and represents the order placed by the customer. This table is linked through the CustomerId column to the Customers table and its Id column. That is, the CustomerId column is a foreign key that points to the Id column from the Customers table.

A foreign key definition at the table level would look like this:

```
1   CREATE TABLE orders
2   (
3   ID INT PRIMARY key IDENTITY,
4   CustomerId INT,
5   CreatedAtDate,
6   FOREIGN key (CustomerId) REFERENCES Customers (Id)
7   );
```

You can use the CONSTRAINT statement to specify a name for a foreign key constraint. Usually this name starts with "FK_" prefix:

```
1 CREATE TABLE orders
2 (
3 ID INT PRIMARY key IDENTITY,
4 CustomerId INT,
5 CreatedAtDate,
6 CONSTRAINT FK_Orders_To_Customers FOREIGN key (CustomerId) REFERENCES
  Customers (Id)
7 );
```

In this case, the foreign key constraint CustomerId is named "FK_Orders_To_Customers".

ON DELETE and ON UPDATE

Using the ON DELETE and ON UPDATE statements, you can set the actions to be taken, respectively, when the associated row is deleted and updated from the main table. And to define an action, we can use the following options:

- **CASCADE**: Automatically removes or modifies rows from a dependent table when related rows in the primary table are deleted or modified.
- **NO ACTION**: Prevents any action on the dependent table when related rows in the master table are deleted or modified. That is, in fact, there are no actions.
- **SET NULL**: When deleting a related row from the main table, sets the foreign key column to NULL.
- **SET DEFAULT**: When deleting a related row from the main table, sets the foreign key column to its default value, which is set using the DEFAULT attribute. If a column does not have a default value, NULL is used as the column value.

  Cascading delete

  By default, if any row from the dependent table refers to a row from the main table by foreign key, then we will not be able to delete this row from the main table. First, we will need to remove all related rows from the dependent table. And if, when deleting a row from the main table, it is necessary that all related rows from the dependent table be deleted, then cascade deletion is applied, that is, the CASCADE option:

```
1   CREATE TABLE orders
2   (
3   ID INT PRIMARY key IDENTITY,
4   CustomerId INT,
5   CreatedAtDate,
6   FOREIGN key (CustomerId) REFERENCES Customers (Id) ON DELETE CASCADE
7   )
```

  The ON UPDATE CASCADE statement works similarly. Changing the value of a primary key will automatically change the value of its associated foreign key. But since primary keys tend to change very infrequently, and it is generally not recommended to use columns with mutable values as primary keys, in practice the ON UPDATE statement is rarely used.

  NULL setting

  Setting the foreign key to SET NULL requires the foreign key column to be nullable:

```
1   CREATE TABLE orders
2   (
3   ID INT PRIMARY key IDENTITY,
```

```
4   CustomerId INT,
5   CreatedAtDate,
6   FOREIGN key (CustomerId) REFERENCES Customers (Id) ON DELETE SET NULL
7   );
```

Setting the default value

```
1 CREATE TABLE orders
2 (
3 ID INT PRIMARY key IDENTITY,
4 CustomerId INT,
5 CreatedAtDate,
6 FOREIGN key (CustomerId) REFERENCES Customers (Id) ON DELETE SET DEFAULT
7 )
```

Changing a table

Perhaps at some point we want to change an existing table. For example, add or remove columns, change the type of columns, add or remove constraints. That is, you will need to change the definition of the table. The ALTER TABLE statement is used to modify tables.

The general formal command syntax is as follows:

```
1   ALTER TABLE table_name [WITH CHECK | WITH NOCHECK]
2   {ADD column_name column_data_type [column_attributes] |
3   DROP COLUMN column_name |
4   ALTER COLUMN column_name column_data_type [NULL|NOT NULL] |
5   ADD [CONSTRAINT] constraint_definition |
6   DROP [CONSTRAINT] constraint_name}
```

Thus, with the help of ALTER TABLE, we can perform a variety of scenarios for changing a table. Let's consider some of them.

Adding a new column

Add a new Address column to the Customers table:

```
1   ALTER TABLE Customers
2   ADD Address NVARCHAR(50) NULL;
```

In this case, the Address column is of type NVARCHAR and has a null attribute defined for it. But what if we need to add a column that should not accept NULL values? If there is data in the table, then the following command will not be executed:

```
1   ALTER TABLE Customers
2   ADD Address NVARCHAR(50) NOT NULL;
```

So in this case the solution is to set the default value via the DEFAULT attribute:

```
1   ALTER TABLE Customers
2   ADD Address NVARCHAR(50) NOT NULL DEFAULT 'Unknown';
```

In this case, if the table already has data, then the value "Unknown" will be added for the Address column.

Deleting a column

Let's remove the Address column from the Customers table:

```
1   ALTER TABLE Customers
2   DROP COLUMN address;
```

Changing the type of a column

Let's change the data type of the FirstName column in the Customers table to NVARCHAR(200):

```
1   ALTER TABLE Customers
2   ALTER COLUMN FirstName NVARCHAR(200);
```

Adding a CHECK constraint

When you add constraints, SQL Server automatically checks the existing data against the constraints you add. If the data does not meet the restrictions, then such restrictions will not be added. For example, let's set the Age column in the Customers table to be constrained to Age > 21.

```
1   ALTER TABLE Customers
2   ADD CHECK (Age > 21);
```

If there are rows in the table that have values in the Age column that do not meet this restriction, then the sql command will fail. To avoid such a check for compliance and still add a constraint despite the presence of non-matching data, use the WITH NOCHECK clause:

```
1   ALTER TABLE Customers WITH NOCHECK
2   ADD CHECK (Age > 21);
```

The default value is WITH CHECK, which checks against the constraints.

Adding a foreign key

Suppose that two tables are initially added to the database, which are not connected in any way:

```
1    CREATE TABLE Customers
2    (
3    ID INT PRIMARY key IDENTITY,
```

```
4    Age INT DEFAULT eighteen,
5    FirstName NVARCHAR(20) NOT NULL,
6    LastName NVARCHAR(20) NOT NULL,
7    Email VARCHAR(30) UNIQUE,
8    Phone VARCHAR(20) UNIQUE
9    );
10   CREATE TABLE orders
11   (
12   ID INT IDENTITY,
13   CustomerId INT,
14   CreatedAtDate
15   );
```

Add a foreign key constraint to the CustomerId column of the Orders table:

```
1   ALTER TABLE orders
2   ADD FOREIGN KEY(CustomerId) REFERENCES Customers(Id);
```

Adding a Primary Key

Using the Orders table defined above, let's add a primary key for the Id column to it:

```
1   ALTER TABLE orders
2   ADD PRIMARY key (id);
```

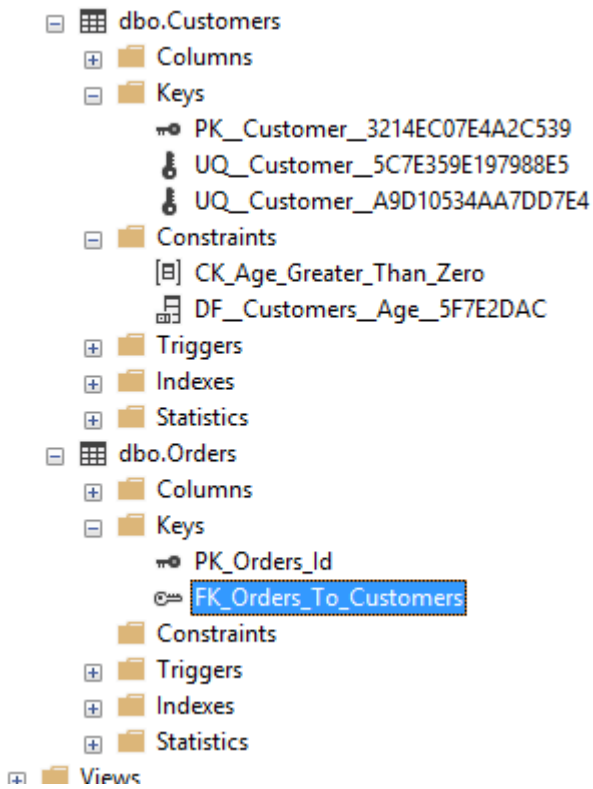
Adding Constraints with Names

When adding constraints, we can specify a name for them using the CONSTRAINT statement followed by the name of the constraint:

```
1 ALTER TABLE orders
2 ADD CONSTRAINT PK_Orders_Id PRIMARY key (Id)
3 CONSTRAINT   FK_Orders_To_Customers   FOREIGN   KEY(CustomerId)   REFERENCES
  Customers(Id);
4
5 ALTER TABLE Customers
6 ADD CONSTRAINT CK_Age_Greater_Than_Zero CHECK (age > 0);
```

Removing restrictions

To remove restrictions, you need to know their name. If we do not know exactly the name of the constraint, then it can be found through SQL Server Management Studio:



Expanding the tables node in the Keys subnode, you can see the names of the primary and foreign key constraints. Foreign key constraint names start with "FK". And in the Constraints subnode, you can find all the CHECK and DEFAULT constraints. CHECK constraints start with "CK" and DEFAULT constraints start with "DF".

For example, as you can see in the screenshot, in my case, the name of the foreign key constraint in the Orders table is called "FK_Orders_To_Customers". Therefore, to remove a foreign key, we can use the following expression:

```
1   ALTER TABLE Orders
2   DROP FK_Orders_To_Customers;
```

Packages. GO command

In the previous cases, a database was created first, and then a table was added to this database using separate SQL commands. But you can immediately combine several commands in one script. In this case, individual sets of commands are called batches. Each batch consists of one or more SQL statements that are executed as a whole. The GO command serves as a signal for the completion of the package and the execution of its expressions.

The point of batching SQL statements is that some statements must succeed before other statements run. For example, when adding tables, we would need to make sure that the database in which we are going to create tables has been created.

For example, let's define the following script:

```
CREATE DATABASE internetstore;
GO

USE internetstore;

CREATE TABLE Customers
(
Id INT PRIMARY KEY IDENTITY,
Age INT DEFAULT 18,
FirstName NVARCHAR(20) NOT NULL,
LastName NVARCHAR(20) NOT NULL,
Email VARCHAR(30) UNIQUE,
Phone VARCHAR(20) UNIQUE
);

CREATE TABLE orders
(
Id INT PRIMARY KEY IDENTITY,
CustomerId INT,
CreatedAt DATE,
FOREIGN KEY (CustomerId) REFERENCES Customers (Id) ON DELETE CASCADE
);
```

First, the internetstore database is created. Then comes the GO command, which signals that the next batch of expressions can be executed. And then the second batch is executed, which adds two tables to it - Customers and Orders.