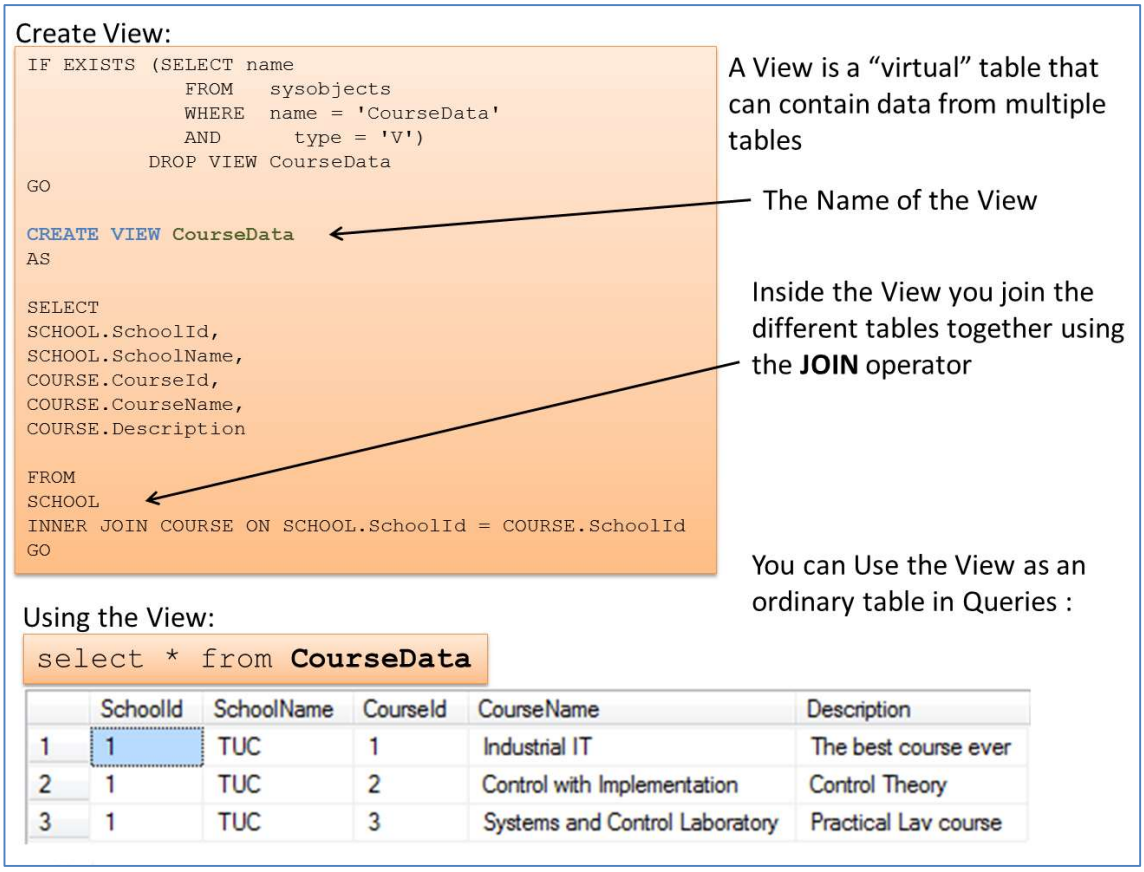
9 Views

Views are virtual table for easier access to data stored in multiple tables.



Syntax for creating a View:

```
CREATE VIEW <ViewName>
AS
…
```

... but it might be easier to do it in the graphical view designer that are built into SQL Management Studio.

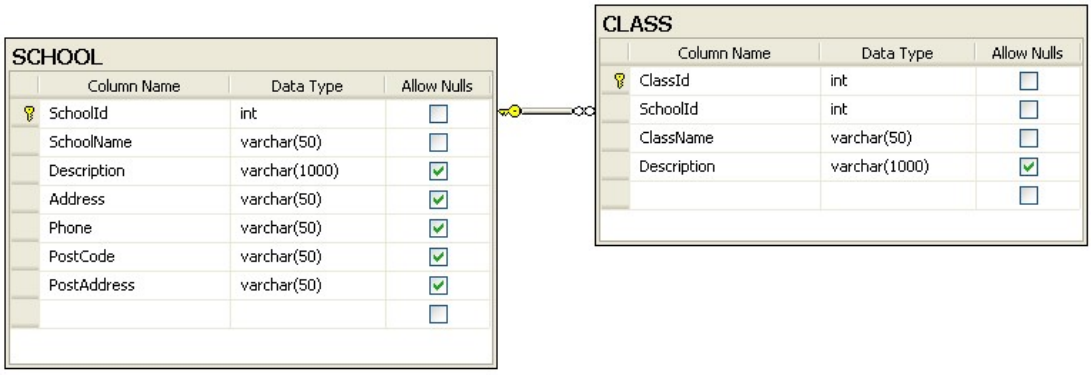
Syntax for using a View:

```
select * from <MyView> where …
```

As shown above, we use a VIEW just like we use an ordinary table.

Example:

We use the SCHOOL and CLASS tables as an example for our View. We want to create a View that lists all the existing schools and the belonging classes.



We create the VIEW using the CREATE VIEW command:

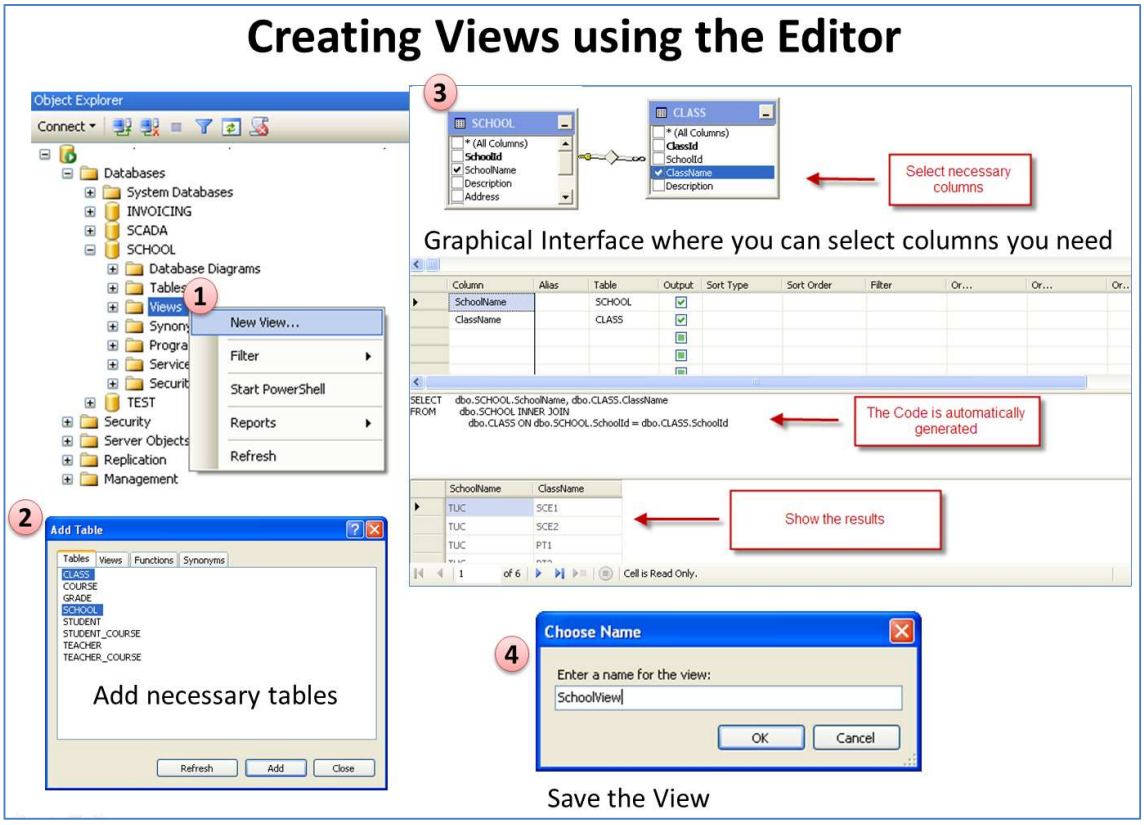
```
CREATE VIEW SchoolView
AS

SELECT
SCHOOL.SchoolName,
CLASS.ClassName
FROM
SCHOOL
INNER JOIN CLASS ON SCHOOL.SchoolId = CLASS.SchoolId
```

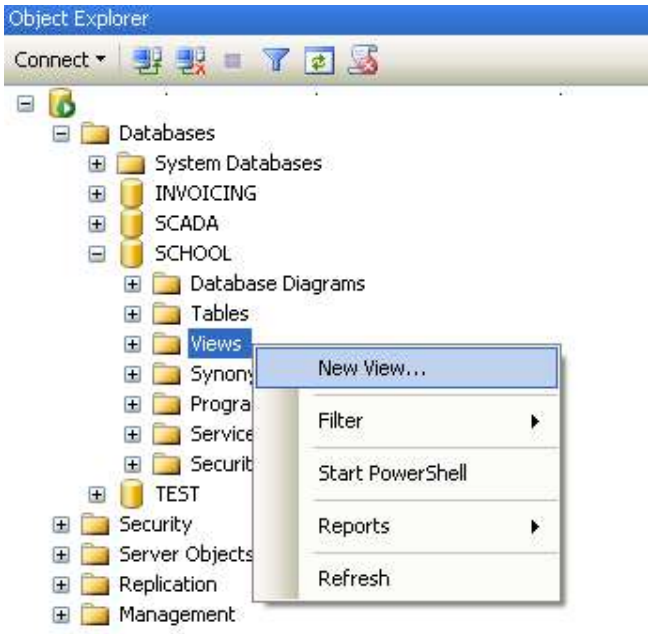
Note! In order to get information from more than one table, we need to link the tables together using a JOIN.

9.1 Using the Graphical Designer

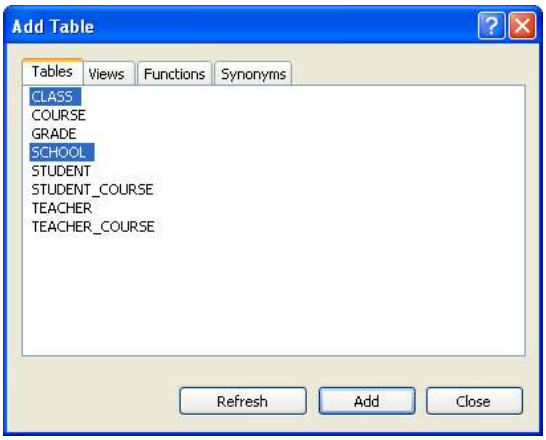
We create the same View using the graphical designer in SQL Server Management Studio:



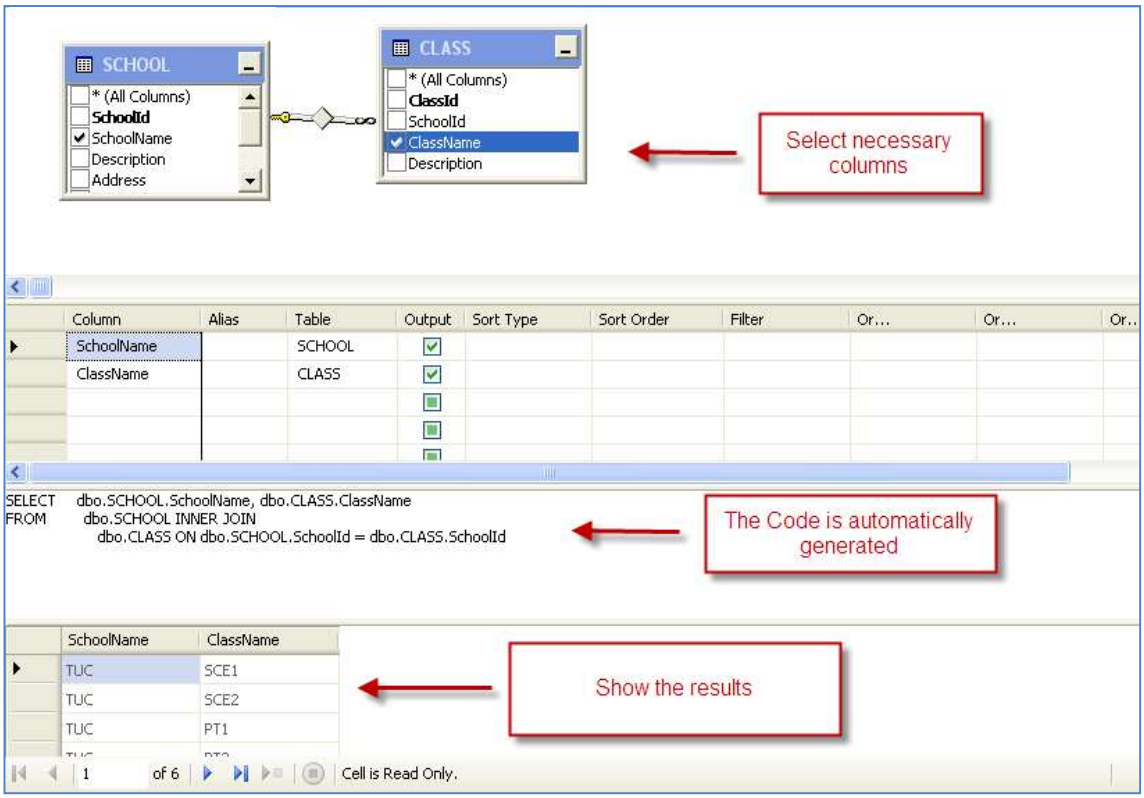
Step 1: Right-click on the View node and select “New View…”:



Step 2: Add necessary tables:



Step 3: Add Columns, etc.



Step 4: Save the VIEW:

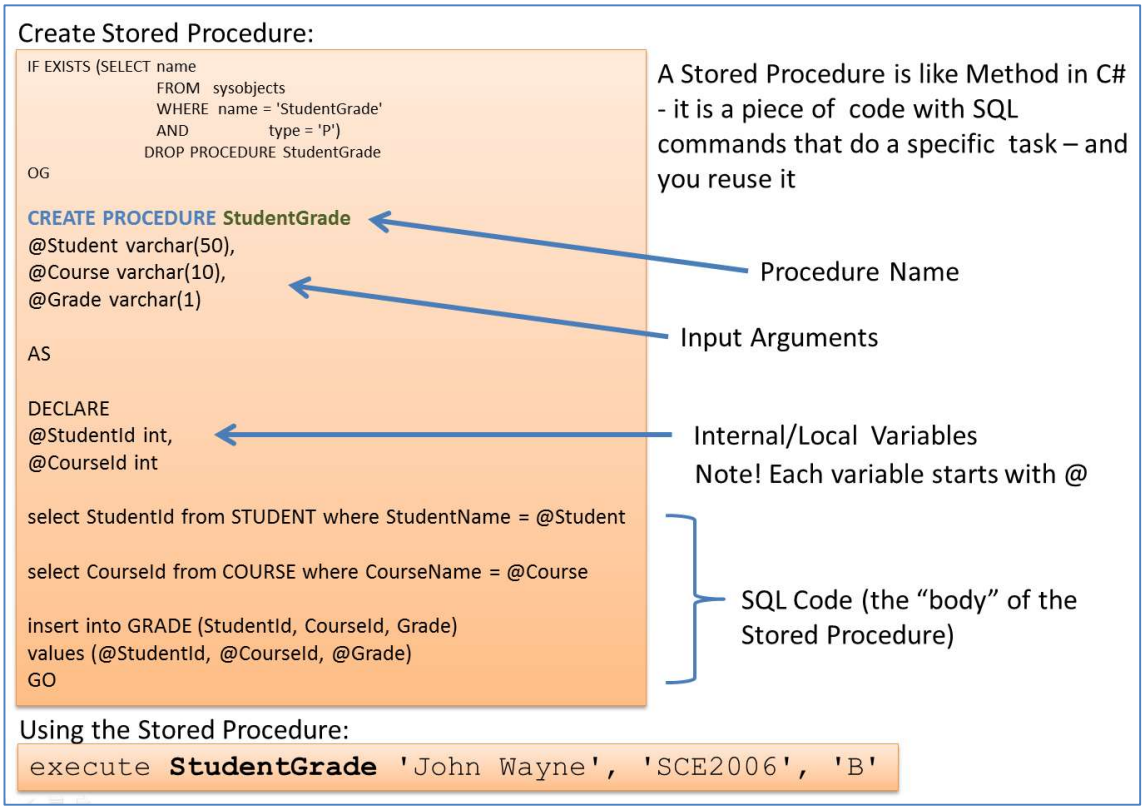
Step 5: Use the VIEW in a query:

```
select * from SchoolView
```

	SchoolName	ClassName
1	TUC	SCE1
2	TUC	SCE2
3	TUC	PT1
4	TUC	PT2
5	NTNU	A1
6	NTNU	A2

10 Stored Procedures

A Stored Procedure is a precompiled collection of SQL statements. In a stored procedure you can use if sentence, declare variables, etc.



Syntax for creating a Stored Procedure:

```
CREATE PROCEDURE <ProcedureName>
@<Parameter1> <datatype>
…
declare
@myVariable <datatype>
… Create your Code here
```

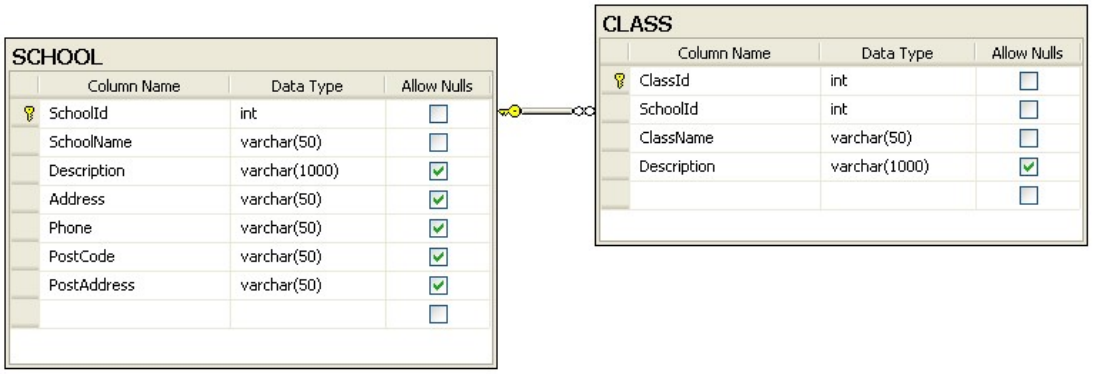
Note! You need to use the symbol "@" before variable names.

Syntax for using a Stored Procedure:

```
EXECUTE <ProcedureName(…)>
```

Example:

We use the SCHOOL and CLASS tables as an example for our Stored Procedure. We want to create a Stored Procedure that lists all the existing schools and the belonging classes.



We create the Stored Procedure as follows:

```
CREATE PROCEDURE GetAllSchoolClasses
AS

select
SCHOOL.SchoolName,
CLASS.ClassName
from
SCHOOL
inner join CLASS on SCHOOL.SchoolId = CLASS.SchoolId
order by SchoolName, ClassName
```

When we have created the Stored Procedure we can run (or execute) the Stored procedure using the execute command like this:

```
execute GetAllSchoolClasses
```

	SchoolName	ClassName
1	NTNU	A1
2	NTNU	A2
3	TUC	PT1
4	TUC	PT2
5	TUC	SCE1
6	TUC	SCE2

We can also create a Store Procedure with input parameters.

Example:

We use the same tables in this example (SCHOOL and CLASS) but now we want to list all classes for a specific school.

The Stored Procedure becomes:

```
CREATE PROCEDURE GetSpecificSchoolClasses
@SchoolName varchar(50)
AS

select
SCHOOL.SchoolName,
CLASS.ClassName
from
SCHOOL
inner join CLASS on SCHOOL.SchoolId = CLASS.SchoolId
where SchoolName=@SchoolName
order by ClassName
```

We run (or execute) the Stored Procedure:

```
execute GetSpecificSchoolClasses 'TUC'
```

	SchoolName	ClassName
1	TUC	PT1
2	TUC	PT2
3	TUC	SCE1
4	TUC	SCE2

or:

```
execute GetSpecificSchoolClasses 'NTNU'
```

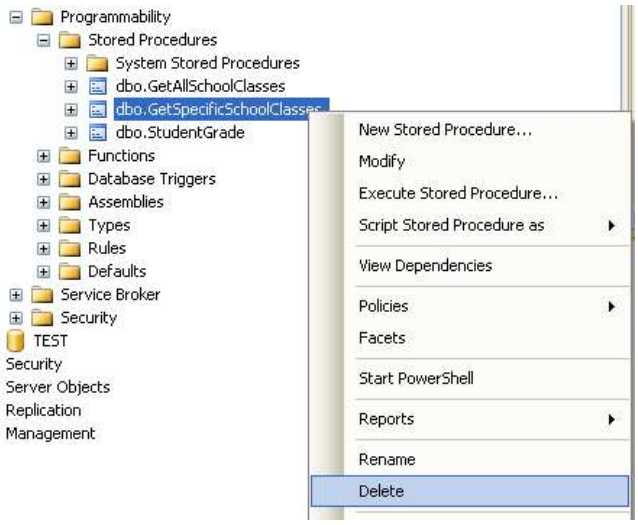
	SchoolName	ClassName
1	NTNU	A1
2	NTNU	A2

When we try to create a Stored Procedure that already exists we get the following error message:

```
There is already an object named 'GetSpecificSchoolClasses' in the database.
```

Then we first need to delete (or DROP) the old Stored Procedure before we can recreate it again.

We can do this manually in the Management Studio in SQL like this:



A better solution is to add code for this in our script, like this:

```
IF EXISTS (SELECT name
        FROM   sysobjects
        WHERE  name = GetSpecificSchoolClasses '
        AND        type = 'P')
     DROP PROCEDURE GetSpecificSchoolClasses
GO

CREATE PROCEDURE GetSpecificSchoolClasses
@SchoolName varchar(50)
AS

select
SCHOOL.SchoolName,
CLASS.ClassName
from
SCHOOL
inner join CLASS on SCHOOL.SchoolId = CLASS.SchoolId
where SchoolName=@SchoolName
order by ClassName
```

So we use CREATE PROCEDURE to create a Stored Procedure and we use DROP PROCEDURE to delete a Stored Procedure.

10.1 NOCOUNT ON/NOCOUNT OFF

In advanced Stored Procedures and Script, performance is very important. Using SET NOCOUNT ON and SET NOCOUNT OFF makes the Stored Procedure run faster.

SET NOCOUNT ON stops the message that shows the count of the number of rows affected by a Transact-SQL statement or stored procedure from being returned as part of the result set.

SET NOCOUNT ON prevents the sending of DONE_IN_PROC messages to the client for each statement in a stored procedure. For stored procedures that contain several statements that do not return much actual data, or for procedures that contain Transact-SQL loops, setting SET NOCOUNT to ON can provide a significant performance boost, because network traffic is greatly reduced.

Example:

```
IF EXISTS (SELECT name
        FROM   sysobjects
        WHERE  name = 'sp_LIMS_IMPORT_REAGENT'
        AND       type = 'P')
      DROP PROCEDURE sp_LIMS_IMPORT_REAGENT
GO

CREATE PROCEDURE sp_LIMS_IMPORT_REAGENT
@Name varchar(100),
@LotNumber varchar(100),
@ProductNumber varchar(100),
@Manufacturer varchar(100)

AS
SET NOCOUNT ON

if not exists (SELECT ReagentId FROM LIMS_REAGENTS WHERE
[Name]=@Name)
      INSERT INTO LIMS_REAGENTS ([Name], ProductNumber, Manufacturer)
      VALUES (@Name, @ProductNumber, @Manufacturer)
else
UPDATE LIMS_REAGENTS SET
      [Name] = @Name,
      ProductNumber = @ProductNumber,
      Manufacturer = @Manufacturer,
      WHERE [Name] = @Name

SET NOCOUNT OFF
GO
```

This Stored Procedure updates a table in the database and in this case you don't normally need feedback, sp setting SET NOCOUNT ON at the top in the stored procedure is a good idea. it is also good practice to SET NOCOUNT OFF at the bottom of the stored procedure.

11 Functions

With SQL and SQL Server you can use lots of built-in functions or you may create your own functions. Here we will learn to use some of the most used built-in functions and in addition we will create our own function.

11.1 Built-in Functions

SQL has many built-in functions for performing calculations on data.

We have 2 categories of functions, namely **aggregate** functions and **scalar** functions. Aggregate functions return a single value, calculated from values in a column, while scalar functions return a single value, based on the input value.

Aggregate functions - examples:

- **AVG()** - Returns the average value
- **STDEV()** - Returns the standard deviation value
- **COUNT()** - Returns the number of rows
- **MAX()** - Returns the largest value
- **MIN()** - Returns the smallest value
- **SUM()** - Returns the sum
- etc.

Scalar functions - examples:

- **UPPER()** - Converts a field to upper case
- **LOWER()** - Converts a field to lower case
- **LEN()** - Returns the length of a text field
- **ROUND()** - Rounds a numeric field to the number of decimals specified
- **GETDATE()** - Returns the current system date and time
- etc.

11.1.1 String Functions

Here are some useful functions used to manipulate with strings in SQL Server:

- CHAR
- CHARINDEX
- REPLACE
- SUBSTRING
- LEN
- REVERSE
- LEFT
- RIGHT
- LOWER
- UPPER
- LTRIM
- RTRIM

Read more about these functions in the SQL Server Help.

11.1.2 Date and Time Functions

Here are some useful Date and Time functions in SQL Server:

- DATEPART
- GETDATE
- DATEADD
- DATEDIFF
- DAY
- MONTH
- YEAR
- ISDATE

Read more about these functions in the SQL Server Help.

11.1.3 Mathematics and Statistics Functions

Here are some useful functions for mathematics and statistics in SQL Server:

- COUNT
- MIN, MAX
- COS, SIN, TAN
- SQRT
- STDEV
- MEAN
- AVG

Read more about these functions in the SQL Server Help.

11.1.4 AVG()

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

Syntax:

```
SELECT AVG(column_name) FROM table_name
```

Example:

Given a GRADE table:

	Column Name	Data Type	Allow Nulls
🔑	GradeId	int	☐
	StudentId	int	☐
	CourseId	int	☐
	Grade	float	☐
	Comment	varchar(1000)	☑

We want to find the average grade for a specific student:

```
select AVG(Grade) as AvgGrade from GRADE where StudentId=1
```

	AvgGrade
1	4,5

11.1.5 COUNT()

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

The COUNT(column_name) function returns the number of values (NULL values will not be counted) of the specified column:

```
SELECT COUNT(column_name) FROM table_name
```

The COUNT(*) function returns the number of records in a table:

```
SELECT COUNT(*) FROM table_name
```

We use the CUSTOMER table as an example:

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111
2	2	1001	Jackson	Smith	45	London	22222222
3	3	1002	Johnsen	John	32	London	33333333

```
select COUNT(*) as NumbersofCustomers from CUSTOMER
```

	NumberofCustomers
1	3

11.1.6 The GROUP BY Statement

Aggregate functions often need an added GROUP BY statement.

The GROUP BY statement is used in conjunction with the aggregate functions to group the result-set by one or more columns.

Syntax

```
SELECT column_name, aggregate_function(column_name)
FROM table_name
WHERE column_name operator value
GROUP BY column_name
```

Example:

We use the CUSTOMER table as an example:

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111
2	2	1001	Jackson	Smith	45	London	22222222
3	3	1002	Johnsen	John	32	London	33333333

If we try the following:

```
select FirstName, MAX(AreaCode) from CUSTOMER
```

We get the following error message:

```
Column 'CUSTOMER.FirstName' is invalid in the select list because it is not contained in
either an aggregate function or the GROUP BY clause.
```

The solution is to use the GROUP BY:

```
select FirstName, MAX(AreaCode) from CUSTOMER
group by FirstName
```

	FirstName	(No column name)
1	John	32
2	Smith	45

11.1.7 The HAVING Clause

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

Syntax:

```
SELECT column_name, aggregate_function(column_name)
FROM table_name
WHERE column_name operator value
GROUP BY column_name
HAVING aggregate_function(column_name) operator value
```

We use the GRADE table as an example:

```
select * from GRADE
```

	GradeId	StudentId	CourseId	Grade	Comment
1	1	1	1	4	NULL
2	2	2	1	5	NULL
3	3	3	3	0	NULL
4	4	4	3	3	NULL
5	5	1	3	5	NULL

First we use the GROUP BY statement:

```
select CourseId, AVG(Grade) from GRADE
group by CourseId
```

	CourseId	(No column name)
1	1	4,5
2	3	2,66666666666667

While the following query:

```
select CourseId, AVG(Grade) from GRADE
group by CourseId
having AVG(Grade)>3
```

	CourseId	(No column name)
1	1	4,5