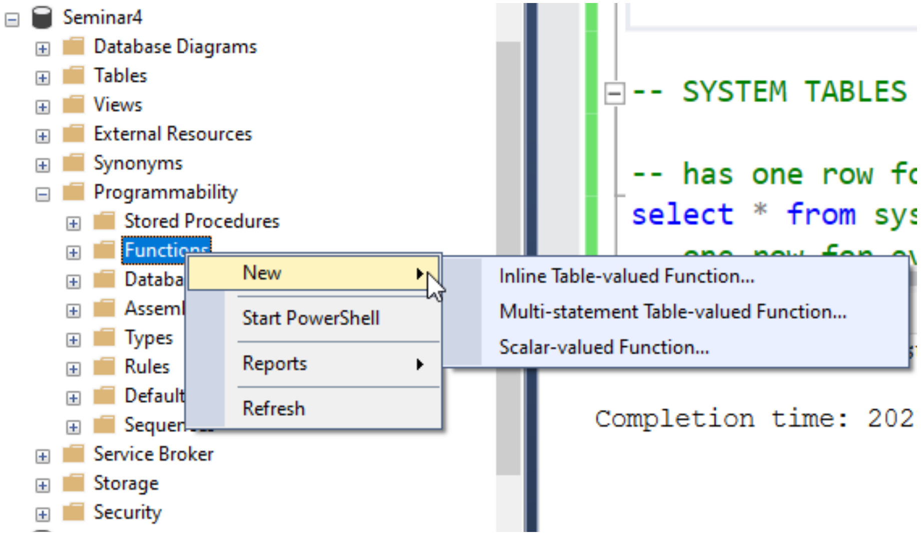
Seminar 4

Functions. Views. System catalog. Triggers. Cursors. Merge statement - SQL Server -

User-Defined Functions

- SQL Server provide the possibility to create functions that can be used after in SQL queries
- The ***user-defined function*** can have input par
- The ***user-defined functions*** are:
  - **Scalar functions**
  - **Inline table-valued functions**
  - **Multi-statement table-valued functions**



```
-- SYSTEM TABLES

-- has one row fc
select * from sys
```

```
Completion time: 202
```

Scalar functions

- Return a scalar value
- When a scalar function is operating on multiple rows, SQL Server executes the function once / row in the result set; this can have a significant impact on performance

User-Defined Functions

Scalar functions

- Create a scalar function:

```
CREATE FUNCTION scalar_function_name(@parameter1 datatype1, @parameter2 datatype2)
RETURNS datatype AS
BEGIN
    -- SQL Statements
RETURN value;
END;
```

- Modify a scalar function:

```
ALTER FUNCTION scalar_function_name(@parameter1 datatype1, @parameter2 datatype2)
RETURNS datatype AS
BEGIN
    -- SQL Statements
RETURN value;
END;
```

- Remove a scalar function:

```
 DROP FUNCTION scalar_function_name;
```

User-Defined Functions

Scalar functions – example

```
CREATE FUNCTION ufNoStudentsPerGroup(@Gid INT)
RETURNS INT AS
BEGIN
	DECLARE @no INT
	SET @no = 0
	SELECT @no= COUNT(*)
	FROM Students
	WHERE Gid = @Gid
	RETURN @no
END
GO

PRINT dbo.ufNoStudentsPerGroup(221) -- 3
PRINT dbo.ufNoStudentsPerGroup(822) -- 2
PRINT dbo.ufNoStudentsPerGroup(821) -- 0
```

- ALTER FUNCTION ufNoStudentsPerGroup ….
- DROP FUNCTION ufNoStudentsPerGroup

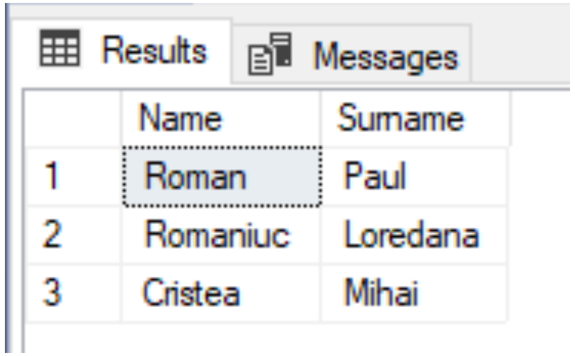
User-Defined Functions

Inline table-valued functions

- Return a table (instead of one value)
- Can be used wherever a table is need it; usually it is called in the FROM clause of a Transact-SQL query
- Contain only ***one Transact-SQL statement***
- The *multi-statement table-valued functions* return also a table, but can contain multiple Transact-SQL statements, despite the *inline table-valued functions* that contain only one Transact-SQL statement

Example:

```
CREATE FUNCTION ufStudentsNamePerGroup(@Gid INT)
RETURNS TABLE
AS
RETURN
    SELECT Name, Surname
    FROM Students
    WHERE Gid= @Gid
GO

SELECT * FROM ufStudentsNamePerGroup(221)
```



	Name	Surname
1	Roman	Paul
2	Romaniuc	Loredana
3	Cristea	Mihai

User-Defined Functions

Multi-statement table-valued functions

- Return a table
- Unlike the *inline table-valued functions*, the *multi-statement table-valued functions* can contain more than one Transact-SQL statement

Example:

```
CREATE FUNCTION ufCoursesWithCredits(@NoOfCredits INT)
RETURNS @CoursesWithCredits TABLE (CoId INT, Title VARCHAR(50), NoOfCredits INT)
AS
BEGIN
    INSERT INTO @CoursesWithCredits
    SELECT CoId, Title, NoOfCredits
    FROM Courses
    WHERE NoOfCredits = @NoOfCredits
    IF @@ROWCOUNT = 0
    INSERT INTO @CoursesWithCredits
    VALUES (0,'No course was found for that number of credits.', 0)
RETURN
END
GO
```

```
-- execute
SELECT * FROM ufCoursesWithCredits(5) -- 12 Operation Systems 5
SELECT * FROM ufCoursesWithCredits(6) -- 11 Databases 6
SELECT * FROM ufCoursesWithCredits(8) -- 0 No course was found ... 0
```

Views

View = a virtual table based on the result set of a query

- Contains columns and records, as a classic table
- The data from the view can be obtained from one or more tables in an alternative manner
- It can have at most 1024 columns
- A view doesn’t store data; a view stores the definition of a query
- Each time a *view* is queried, the database will (re)create the data by using the **SELECT** statement specified in the **CREATE VIEW**, such that a view will have all the time, the data updated
- The name of the columns from a view must be ***unique*** (for the name of the columns that have the same name and are from diferent tables, it is going to be used the **alias** for one of them)
- Create view syntax:

```
CREATE VIEW view_name
AS SELECT_statement
```

- Modify view syntax:

```
ALTER VIEW view_name
AS SELECT_statement
```

- Remove view syntax:

```
DROP VIEW view_name
```

Views

- A view ***cannot use*** the clause **ORDER BY** in the definition (except if in the definition of the view is the clause **TOP**, **OFFSET** or **FOR XML**)
- The records from the result set returned from the view can be order by using the **ORDER BY** clause in the ***execution / quering / SELECT of the view***
- To display the definition of a view can be used the function **OBJECT_DEFINITION** or the stored procedure **sp_helptext**

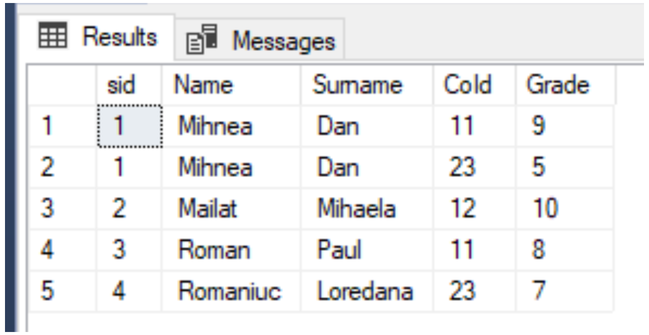
```
PRINT OBJECT_DEFINITION (OBJECT_ID('schema_name.view_name’);
EXEC sp_helptext    'schema_name.view_name’;
```

- In a view can be inserted records only if the insert affects only one base table (if the view contains data from multiple tables)
- In a view can be modified the records only if the insert affects only one base table (if the view contains data from multiple tables)
- From a view can be deleted records only if the view contains data from just one table
- The insert operations in a view are possible only if the view expose all the column that are not allowing the value NULL

Views

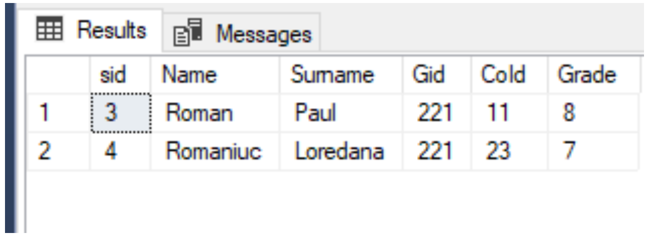
View example:

```
-- create / alter view
CREATE OR ALTER VIEW vSEC
AS
        SELECT S.Sid, S.Name, S.Surname, C.CoId, E.Grade
        FROM Students S INNER JOIN Exams E ON S.Sid= E.Sid
        INNER JOIN Courses C ON E.CoId = C.CoId
GO

SELECT * FROM vSEC
```



	sid	Name	Surname	CoId	Grade
1	1	Mihnea	Dan	11	9
2	1	Mihnea	Dan	23	5
3	2	Mailat	Mihaela	12	10
4	3	Roman	Paul	11	8
5	4	Romaniuc	Loredana	23	7

```
-- alter view
ALTER VIEW vSEC
AS
        SELECT S.Sid, S.Name, S.Surname, S.Gid, C.CoId, E.Grade
        FROM Students S INNER JOIN Exams E ON S.Sid= E.Sid
        INNER JOIN Courses C ON E.CoId = C.CoId
        WHERE S.Gid=221
GO

SELECT Sid, Name, Surname, Gid, CoId, Grade FROM vSEC
```

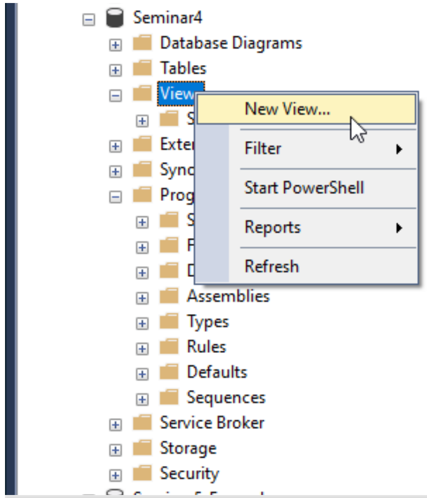


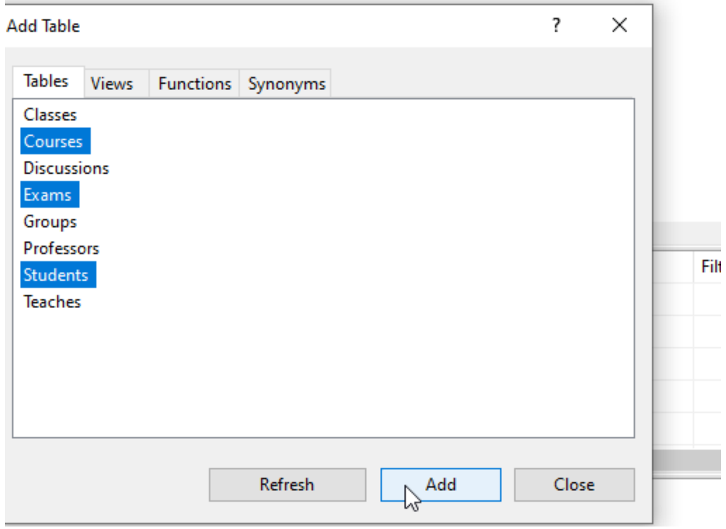
	sid	Name	Surname	Gid	CoId	Grade
1	3	Roman	Paul	221	11	8
2	4	Romaniuc	Loredana	221	23	7

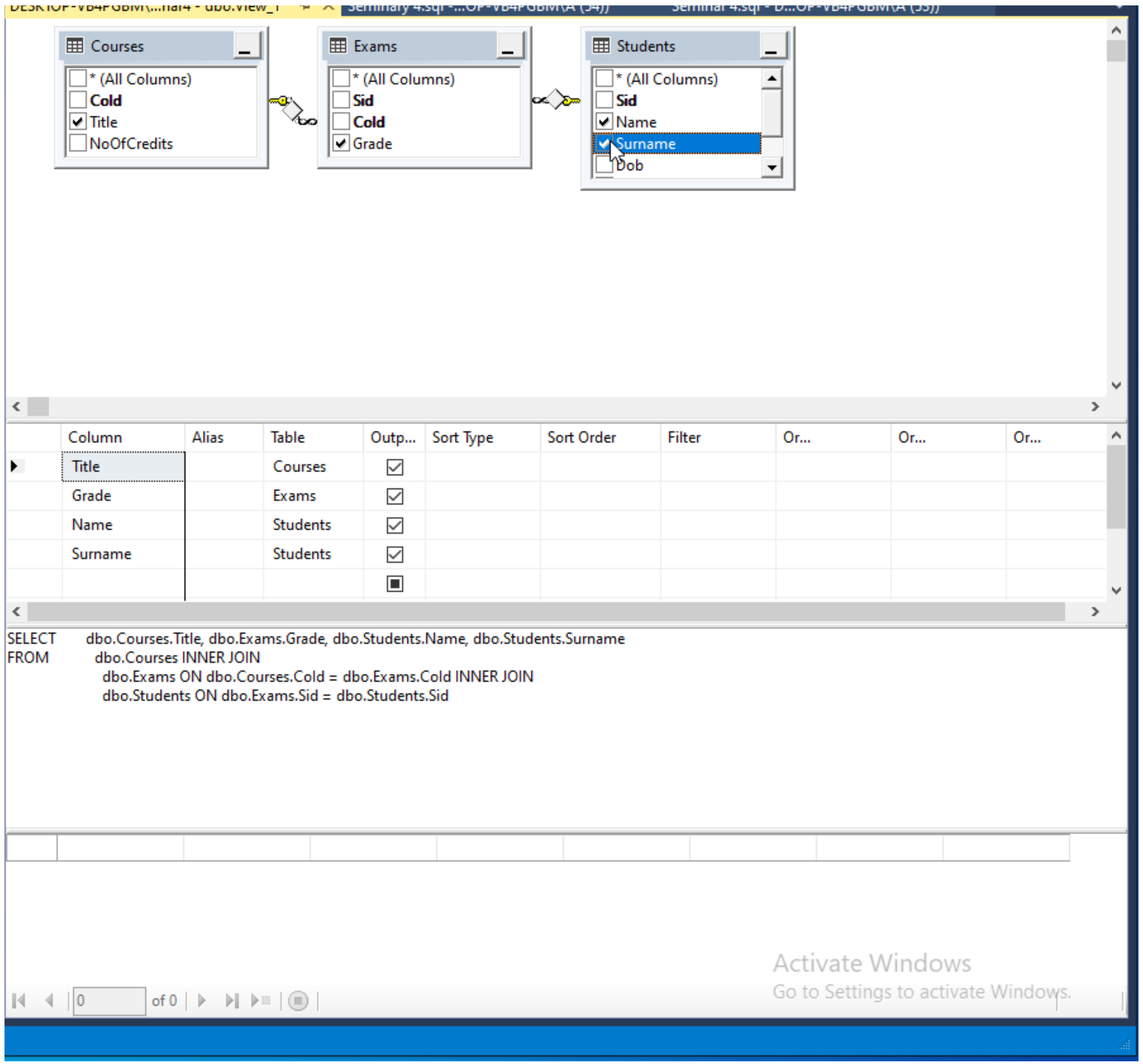
```
-- remove view
DROP VIEW vSEC
```

Views

Create view by Design View





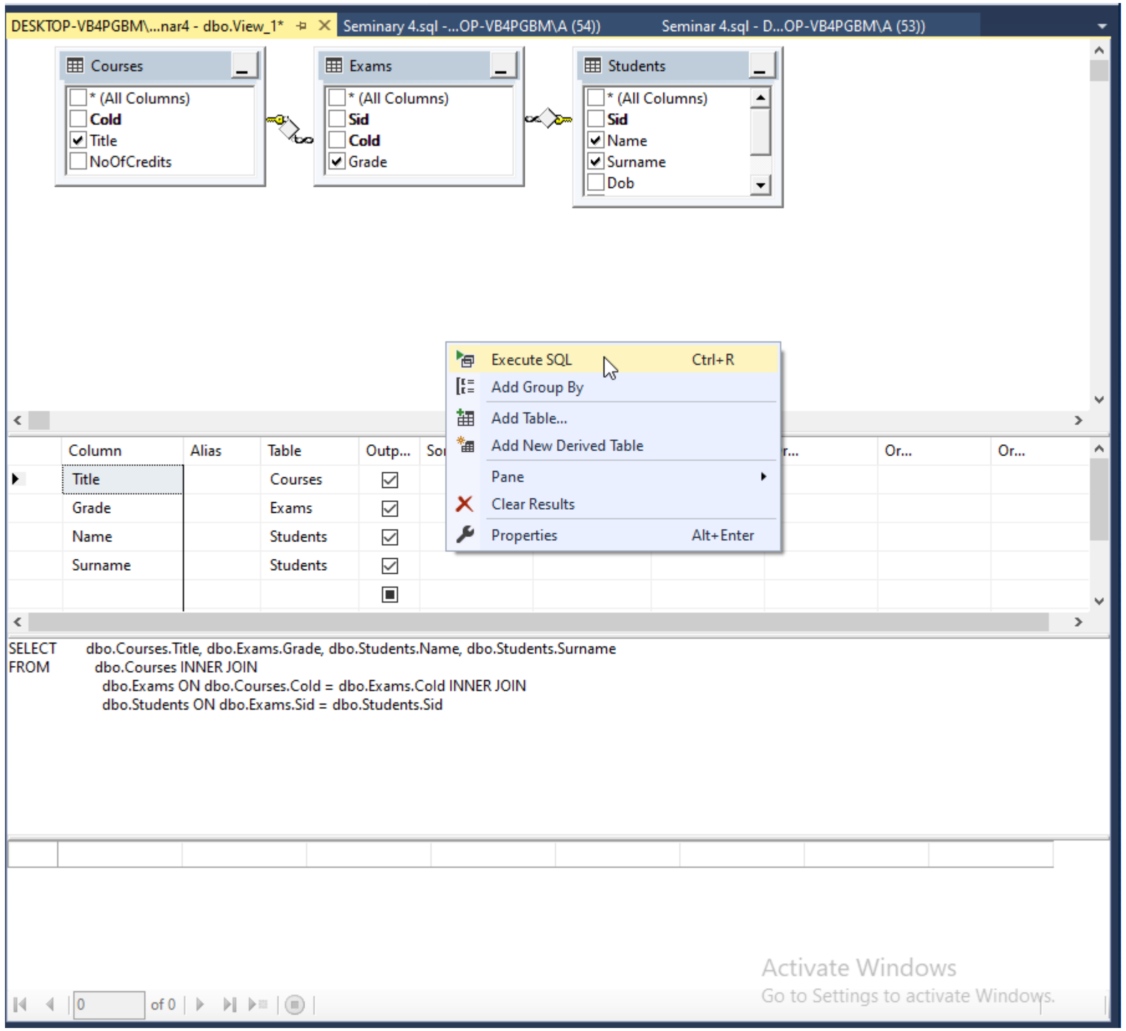


Column	Alias	Table	Outp...	Sort Type	Sort Order	Filter	Or...	Or...	Or...
Title		Courses	☑						
Grade		Exams	☑						
Name		Students	☑						
Surname		Students	☑						
			■						

```
SELECT    dbo.Courses.Title, dbo.Exams.Grade, dbo.Students.Name, dbo.Students.Surname
FROM      dbo.Courses INNER JOIN
            dbo.Exams ON dbo.Courses.Cold = dbo.Exams.Cold INNER JOIN
            dbo.Students ON dbo.Exams.Sid = dbo.Students.Sid
```

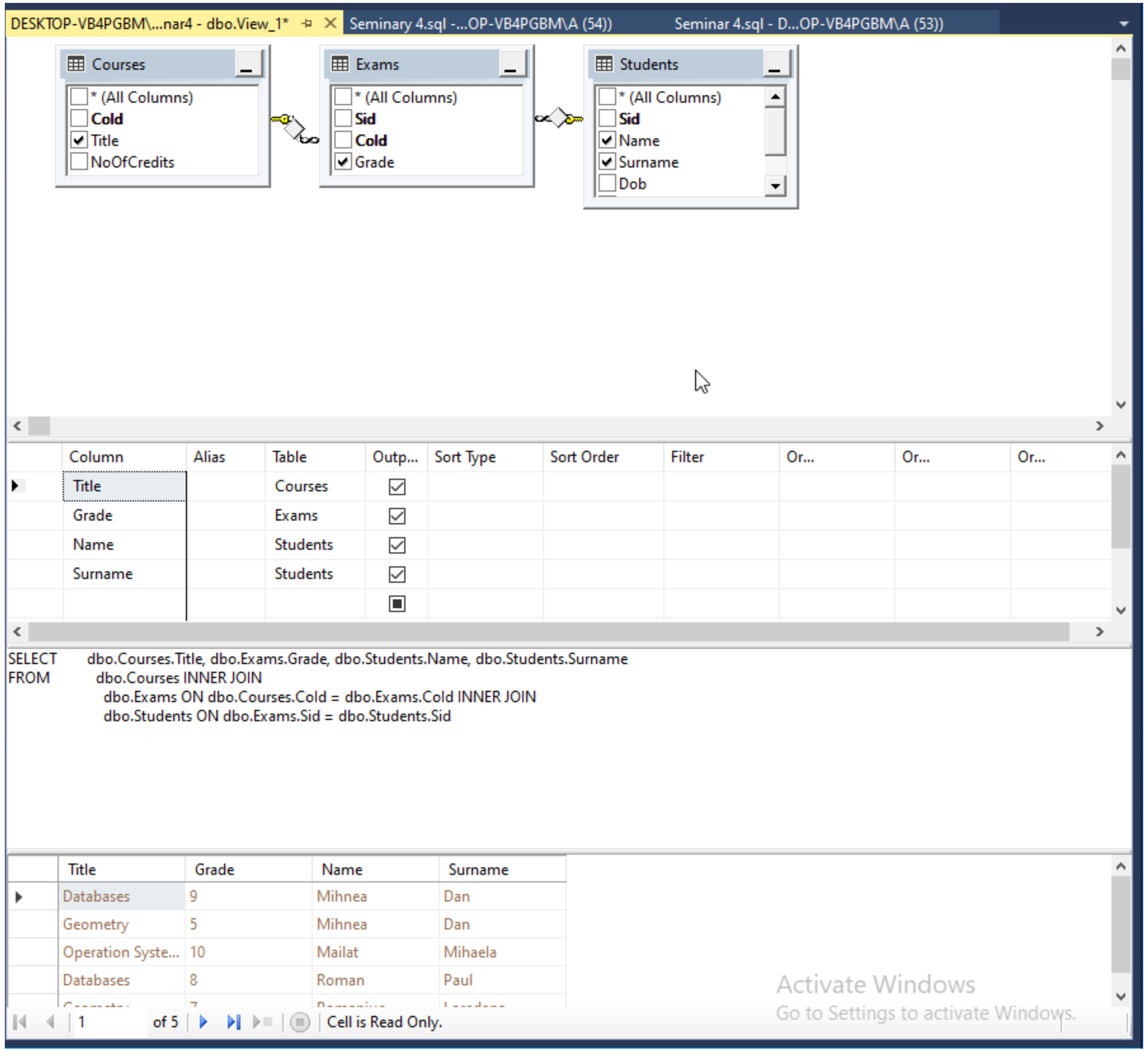
Views

Create view by Design View



Column	Alias	Table	Outp...	So...
Title		Courses	☑	
Grade		Exams	☑	
Name		Students	☑	
Surname		Students	☑	

```
SELECT    dbo.Courses.Title, dbo.Exams.Grade, dbo.Students.Name, dbo.Students.Surname
FROM        dbo.Courses INNER JOIN
              dbo.Exams ON dbo.Courses.Cold = dbo.Exams.Cold INNER JOIN
              dbo.Students ON dbo.Exams.Sid = dbo.Students.Sid
```



Column	Alias	Table	Outp...	Sort Type	Sort Order	Filter	Or...	Or...	Or...
Title		Courses	☑						
Grade		Exams	☑						
Name		Students	☑						
Surname		Students	☑						

```
SELECT    dbo.Courses.Title, dbo.Exams.Grade, dbo.Students.Name, dbo.Students.Surname
FROM        dbo.Courses INNER JOIN
              dbo.Exams ON dbo.Courses.Cold = dbo.Exams.Cold INNER JOIN
              dbo.Students ON dbo.Exams.Sid = dbo.Students.Sid
```

Title	Grade	Name	Surname
Databases	9	Mihnea	Dan
Geometry	5	Mihnea	Dan
Operation Syste...	10	Mailat	Mihaela
Databases	8	Roman	Paul

1 of 5 Cell is Read Only.

System Catalog

- Stores data about all the objects from the database (tables, columns, indexes, stored procedures, user-defined functions, views, triggers, …)
- The system catalog is managed by the server (they are not modified directly by the user)

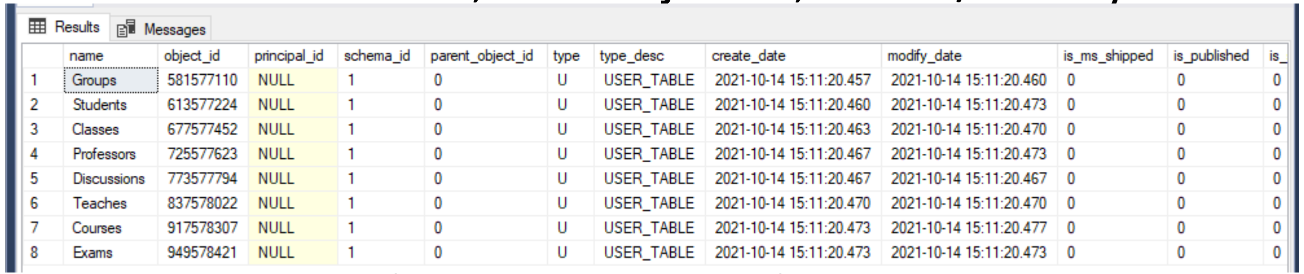
select * from **sys.objects** - has one row for every object from the database (constraint, stored procedure, table, etc) created in the database

select * from **sys.columns** – has one row for every column of an object that has columns (e.g. tables, views, user-defined function that returns a table)

select * from **sys.sql_modules** – has one row for every object that is a module defined in the SQL language in SQL Server (e.g. objects like procedures, functions, … have an associated SQL module).

Example: all the tables from the actual database, with object id, create/modify date

select * from **sys.tables**



	name	object_id	principal_id	schema_id	parent_object_id	type	type_desc	create_date	modify_date	is_ms_shipped	is_published	is_
1	Groups	581577110	NULL	1	0	U	USER_TABLE	2021-10-14 15:11:20.457	2021-10-14 15:11:20.460	0	0	0
2	Students	613577224	NULL	1	0	U	USER_TABLE	2021-10-14 15:11:20.460	2021-10-14 15:11:20.473	0	0	0
3	Classes	677577452	NULL	1	0	U	USER_TABLE	2021-10-14 15:11:20.463	2021-10-14 15:11:20.470	0	0	0
4	Professors	725577623	NULL	1	0	U	USER_TABLE	2021-10-14 15:11:20.467	2021-10-14 15:11:20.473	0	0	0
5	Discussions	773577794	NULL	1	0	U	USER_TABLE	2021-10-14 15:11:20.467	2021-10-14 15:11:20.467	0	0	0
6	Teaches	837578022	NULL	1	0	U	USER_TABLE	2021-10-14 15:11:20.470	2021-10-14 15:11:20.470	0	0	0
7	Courses	917578307	NULL	1	0	U	USER_TABLE	2021-10-14 15:11:20.473	2021-10-14 15:11:20.477	0	0	0
8	Exams	949578421	NULL	1	0	U	USER_TABLE	2021-10-14 15:11:20.473	2021-10-14 15:11:20.473	0	0	0

Triggers

Trigger = special type of stored procedure that is executed automatically when a DDL (CREATE, ALTER, DROP) or DML (INSERT, UPDATE, DELETE) event occurs

- Cannot be executed directly
- Each DML trigger belongs to a single table

Syntax for INSERT / UPDATE / DELETE trigger on table / view:

```
CREATE TRIGGER trigger_name
ON { table | view}
[ WITH <dml_trigger_option> [ ,...n ] ]
{  FOR | AFTER | INSTEAD OF }
{  [INSERT] [,] [UPDATE] [,] [DELETE] }
[ WITH APPEND ]
[ NOT FOR REPLICATION ]
AS { sql_statement [;] [ ,...n ] |
EXTERNAL NAME <method specifier[;] > }
```

Triggers

The moment in which a trigger is executed is specified through one of the options:

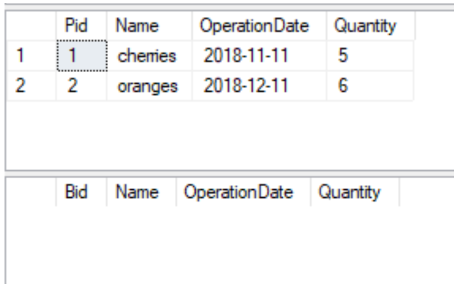
- **FOR**, **AFTER** – this DML trigger is fired only when all the operations specified in the triggering statement have launched successfully (multiple triggers of this type can be defined)
- **INSTEAD OF** – this DML trigger is executed instead of the triggers statement

- When multiple triggers are defined on the same action / event, they are executed in a random order
- When a trigger is executed, 2 special tables can be accessed: ***inserted*** and ***deleted***

Example:

```
create table Product(Pid int primary key identity, Name varchar(50), OperationDate date, Quantity int)
insert into Product Values ('cherries', '2018-11-11', 5), ('oranges', '2018-12-11', 6)
create table BuyLog(Bid int primary key identity, Name varchar(50), OperationDate date, Quantity int)
create table SellLog(Sid int primary key identity, Name varchar(50), OperationDate datetime, Quantity int)
```

```
select * from Product
select * from BuyLog
```



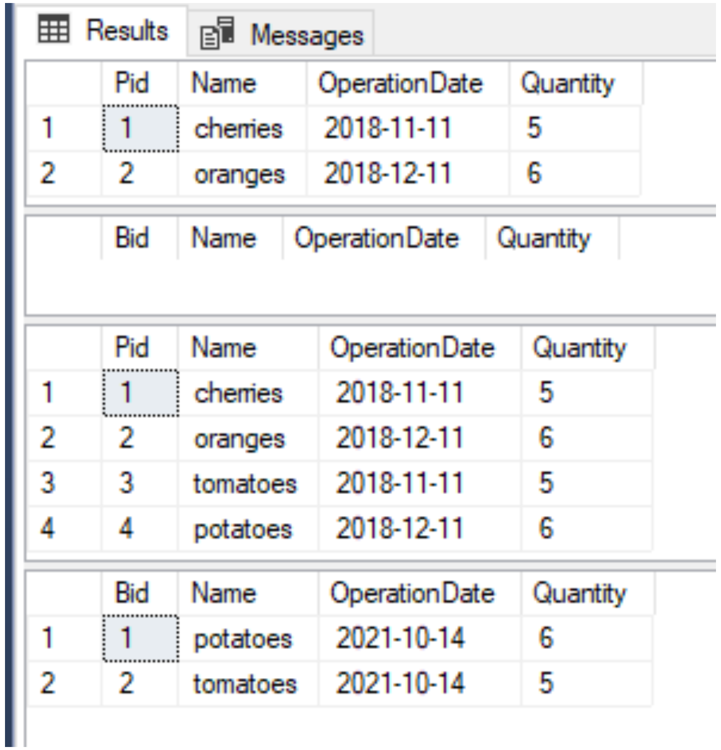
	Pid	Name	OperationDate	Quantity
1	1	cherries	2018-11-11	5
2	2	oranges	2018-12-11	6

	Bid	Name	OperationDate	Quantity

Triggers

Examples: On INSERT

```
CREATE TRIGGER Add_Product
ON Product
FOR INSERT
AS
BEGIN
    INSERT INTO BuyLog (Name, OperationDate, Quantity)
    SELECT Name, GETDATE(), Quantity
    FROM inserted
END
GO
```



	Pid	Name	OperationDate	Quantity
1	1	cherries	2018-11-11	5
2	2	oranges	2018-12-11	6

	Bid	Name	OperationDate	Quantity

	Pid	Name	OperationDate	Quantity
1	1	cherries	2018-11-11	5
2	2	oranges	2018-12-11	6
3	3	tomatoes	2018-11-11	5
4	4	potatoes	2018-12-11	6

	Bid	Name	OperationDate	Quantity
1	1	potatoes	2021-10-14	6
2	2	tomatoes	2021-10-14	5

```
select * from Product
select * from BuyLog
        insert into Product Values ('tomatoes', '2018-11-11', 5), ('potatoes', '2018-12-11', 6)
select * from Product
select * from BuyLog
-- after create trigger - obtain inserted values in table Product and also in table BuyLog
```

Triggers

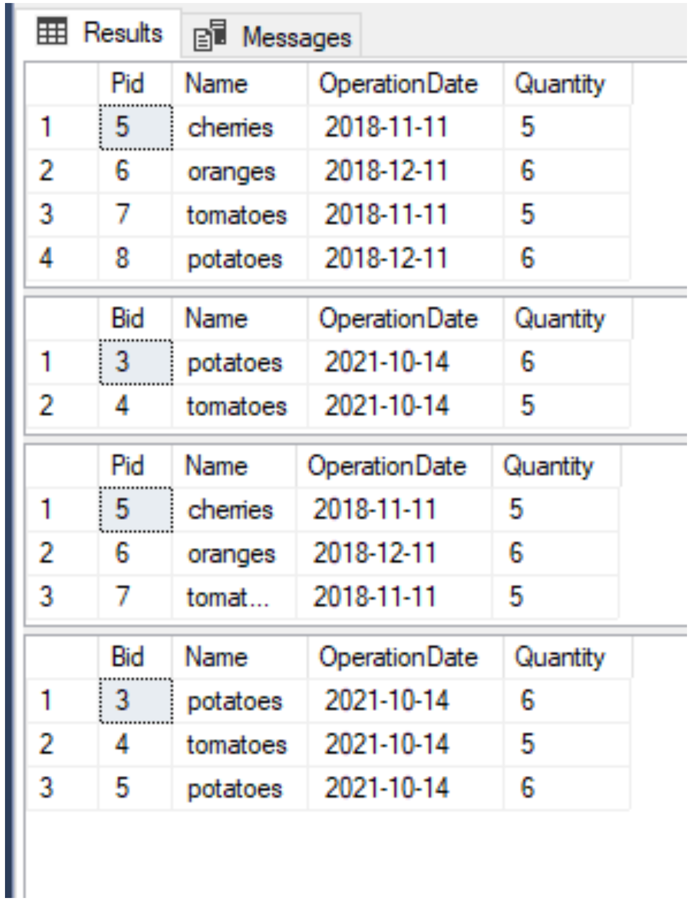
Example: On DELETE

```
        CREATE TRIGGER Delete_Product
        ON Product
        FOR DELETE
        AS
        BEGIN
                SET NOCOUNT ON;
                INSERT INTO BuyLog(Name, OperationDate, Quantity)
                SELECT Name, GETDATE(), Quantity
                FROM deleted
        END
        GO
```

```
select * from Product
select * from BuyLog
                delete from Product where Quantity=6 and Name LIKE 'p%'
```

-- the rows with *potatoes* are deleted from Product and introduced in BuyLog

```
select * from Product
select * from BuyLog
```



	Pid	Name	OperationDate	Quantity
1	5	cherries	2018-11-11	5
2	6	oranges	2018-12-11	6
3	7	tomatoes	2018-11-11	5
4	8	potatoes	2018-12-11	6

	Bid	Name	OperationDate	Quantity
1	3	potatoes	2021-10-14	6
2	4	tomatoes	2021-10-14	5

	Pid	Name	OperationDate	Quantity
1	5	cherries	2018-11-11	5
2	6	oranges	2018-12-11	6
3	7	tomat...	2018-11-11	5

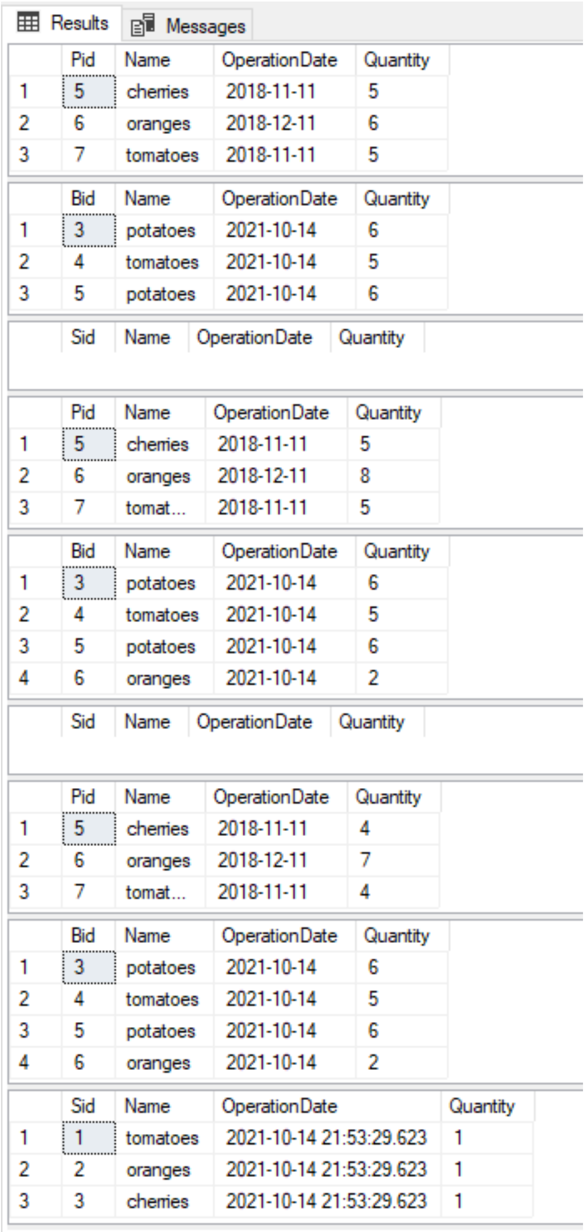
	Bid	Name	OperationDate	Quantity
1	3	potatoes	2021-10-14	6
2	4	tomatoes	2021-10-14	5
3	5	potatoes	2021-10-14	6

Triggers

Example: On UPDATE

```
CREATE TRIGGER Modify_Update_Product
ON Product
FOR UPDATE
AS
BEGIN
	SET NOCOUNT ON;
	INSERT INTO SellLog(Name, OperationDate, Quantity)
	SELECT d.Name, GETDATE(), d.Quantity - i.Quantity
	FROM deleted d INNER JOIN inserted i ON d.Pid = i.Pid
	WHERE i.Quantity < d.Quantity
	INSERT INTO BuyLog(Name, OperationDate, Quantity)
	SELECT i.Name, GETDATE(), i.Quantity - d.Quantity
	FROM deleted d INNER JOIN inserted i ON d.Pid = i.Pid
	WHERE i.Quantity > d.Quantity
END
GO
```

```
select * from Product
select * from BuyLog
select * from SellLog

---- in BuyLog
update Product
set Quantity=8
WHERE Quantity=6

select * from Product
select * from BuyLog
select * from SellLog

--- in SellLog
update Product
set Quantity=Quantity-1
where Name LIKE '%es'

select * from Product
select * from BuyLog
select * from SellLog
```



	Pid	Name	OperationDate	Quantity
1	5	cherries	2018-11-11	5
2	6	oranges	2018-12-11	6
3	7	tomatoes	2018-11-11	5

	Bid	Name	OperationDate	Quantity
1	3	potatoes	2021-10-14	6
2	4	tomatoes	2021-10-14	5
3	5	potatoes	2021-10-14	6

	Sid	Name	OperationDate	Quantity

	Pid	Name	OperationDate	Quantity
1	5	cherries	2018-11-11	5
2	6	oranges	2018-12-11	8
3	7	tomat...	2018-11-11	5

	Bid	Name	OperationDate	Quantity
1	3	potatoes	2021-10-14	6
2	4	tomatoes	2021-10-14	5
3	5	potatoes	2021-10-14	6
4	6	oranges	2021-10-14	2

	Sid	Name	OperationDate	Quantity

	Pid	Name	OperationDate	Quantity
1	5	cherries	2018-11-11	4
2	6	oranges	2018-12-11	7
3	7	tomat...	2018-11-11	4

	Bid	Name	OperationDate	Quantity
1	3	potatoes	2021-10-14	6
2	4	tomatoes	2021-10-14	5
3	5	potatoes	2021-10-14	6
4	6	oranges	2021-10-14	2

	Sid	Name	OperationDate	Quantity
1	1	tomatoes	2021-10-14 21:53:29.623	1
2	2	oranges	2021-10-14 21:53:29.623	1
3	3	cherries	2021-10-14 21:53:29.623	1

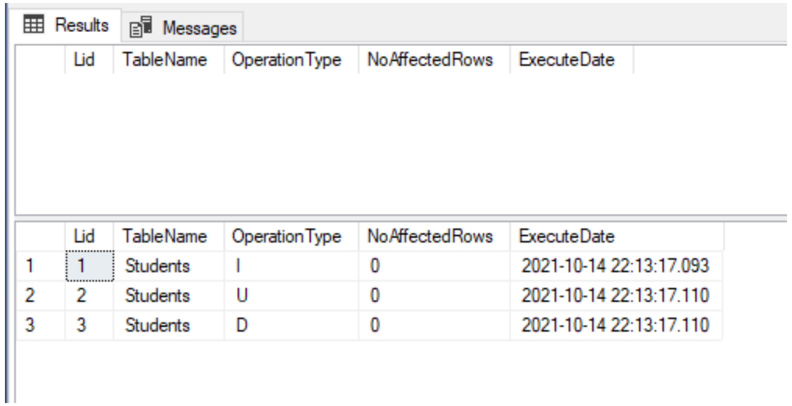
Triggers

Example: On INSERT, UPDATE, UPDATE

CREATE TABLE Log(Lid INT PRIMARY KEY IDENTITY, TableName VARCHAR(30), OperationType CHAR(1), NoAffectedRows INT, ExecuteDate DATETIME)

```
/* Log any operations performed on Students table */
CREATE TRIGGER ChangeStudents ON Students
AFTER INSERT, UPDATE, DELETE
AS
BEGIN
   DECLARE @operation CHAR(1)
   SET @operation = CASE
   WHEN EXISTS(SELECT * FROM inserted) AND EXISTS(SELECT * FROM deleted)
               THEN 'U’
   WHEN EXISTS(SELECT * FROM inserted)
               THEN 'I’
   WHEN EXISTS(SELECT * FROM deleted)
               THEN 'D’
    END
    IF @operation IS NULL
               RETURN
    INSERT INTO Log(TableName, OperationType, NoAffectedRows, ExecuteDate)
    SELECT 'Students', @operation, @@ROWCOUNT, GETDATE()
END
GO
```

```
SELECT * FROM Students
SELECT * FROM Log
INSERT INTO Students VALUES (7, 'Lupea', 'Cristi', '12/12/2002', 'test@a.ro', 822)
UPDATE Students SET Surname = 'Paul' WHERE Name='Lupea'
DELETE FROM Students WHERE Name='Lupea'
SELECT * FROM Log
```



	Lid	TableName	OperationType	NoAffectedRows	ExecuteDate

	Lid	TableName	OperationType	NoAffectedRows	ExecuteDate
1	1	Students	I	0	2021-10-14 22:13:17.093
2	2	Students	U	0	2021-10-14 22:13:17.110
3	3	Students	D	0	2021-10-14 22:13:17.110

Cursors

- There are cases in which the result set should be processed ***row by row*** – can be done with the help of a ***cursor*** opened on a result set
- Opening a cursor on a result set allows the processing of each record from the result set (at one moment one record is processed)
- ***Cursors*** can be used in Transact-SQL scripts, stored procedures, triggers
- When the set-based processing is possible is preferable than to a cursor
- ***Cursors*** extend the processing of the results through:
    - Allow possitioning on specified records from the result set
    - Return a record or a group of records from the current position from the result set
    - Allow modifications on the records from the current position from the result set
    - Support different view levels on the modifications performed by other users on the data from the database that is part of the result set
    - Allow the Transact-SQL statements (from scripts, stored procedures and triggers) access the data from the result set

Cursors

- Transact-SQL statements to declare and populate a cursor, and also to retrieve data:
  - **DECLARE CURSOR** – used to declare the cursor; it specifies a ***SELECT*** statement that will produce the result set
  - **OPEN** - used to populate the cursor; it executes the ***SELECT*** statement from the DECLARE CURSOR statement
  - **FETCH** – fetches individual rows from the result set; usually it is executed multiple times (at least once for every row from the result set)
  - If it is necessarily, an UPDATE or a DELETE statement can be used to modify the row (this step is optional)
  - **CLOSE** - close the cursor and frees the resources (e.g. the result set, the locks from the current record)
  - The cursor is still declared, and so, the OPEN statement can be used to reopen it
  - **DEALLOCATE** – frees all the resources allocated to the cursor from the current session, including its name (after deallocation, the DECLARE statement must be used to rebuild the cursor)

Cursors

- The cursors that are inside of a stored procedure, don’t need to be closed and deallocated; those instructions are executed automatically when the stored procedure close its execution
- The Transact-SQL cursors are extremely efficient when are included in stored procedures and triggers, because all is compiled in a single execution plan on the server, so there is no traffic in the network associated with the returning of the records
- The operation that returns a record from a cursor is called **FETCH**
- The Transact-SQL cursors are using **FETCH** to return records from the result set of a cursor
- Once a cursor is positioned on a row, different operations can be performed on that row
- **FETCH options** to return specified records:
  - **FETCH FIRST** – returns the first row from the cursor
  - **FETCH NEXT** – returns the row immediately after the current row
  - **FETCH PRIOR** – returns the row that is before the current row
  - **FETCH LAST** – returns the last row from the cursor

Cursors

- **FETCH options** to return specified records:
  - **FETCH ABSOLUTE n**, n integer number – returns:
    - n>0: the nth row starting with the first row from the cursor
    - n<0: the nth row before the last row from the cursor
    - n=0: no row
  - **FETCH RELATIVE n**, n integer number - returns:
    - n>0: the row that is *n* rows after the current row
    - n<0: the row that is *n* rows before the current row
    - n=0: no row
- The behaviour of a cursor can be specified in 2 ways:
  - By using the keywords **SCROLL** and **INSENSITIVE** in the **DECLARE CURSOR** statement (in SQL-92 standard)
  - By using the types of cursors
    - Due to the API's for the Databases that define the behaviour of the cursors, the 4 types of cursors are: ***forward-only**, **static*** (or, ***snapshot***, or, ***insensitive***), ***keyset-driven**, **dynamic***

Cursors

- Declare a cursor: Syntax ISO

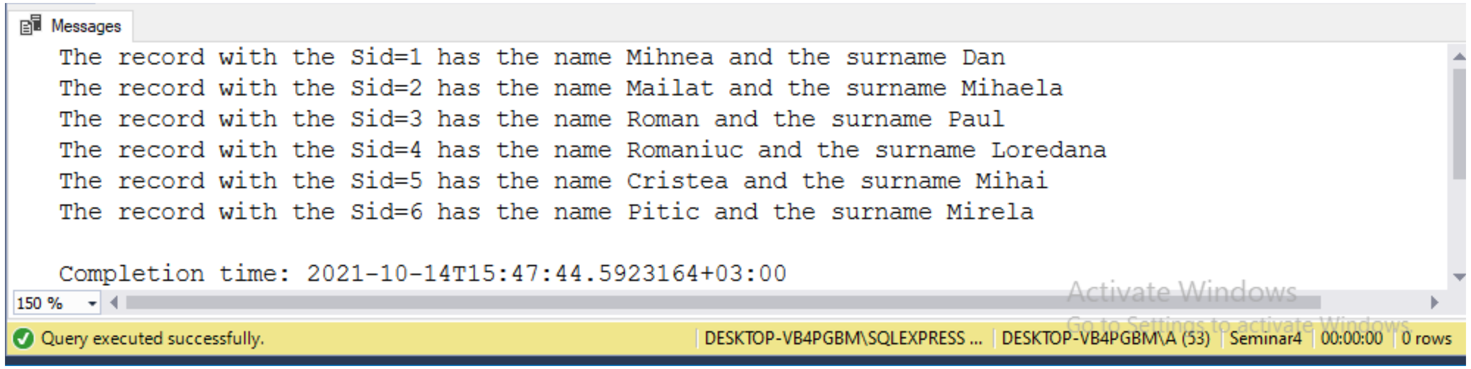
```
DECLARE cursor_name [ INSENSITIVE ] [ SCROLL ] CURSOR
FOR select_statement
[ FOR { READ ONLY | UPDATE [ OF column_name [ ,...n ] ]} ]
[;]
```

- Declare a cursor: Transact-SQL syntax

```
DECLARE cursor_name CURSOR [ LOCAL | GLOBAL ]
[ FORWARD_ONLY | SCROLL ]
[ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
[ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
[ TYPE_WARNING ]
FOR select_statement
[ FOR UPDATE [ OF column_name [ ,...n ] ] ]
[;]
```

Cursors – example

```
DECLARE @Sid INT, @Name VARCHAR(50), @Surname VARCHAR(50)
DECLARE CursorStudents CURSOR FOR
SELECT Sid, Name, Surname
FROM Students
OPEN CursorStudents
FETCH CursorStudents
INTO @Sid, @Name, @Surname
WHILE @@FETCH_STATUS = 0
BEGIN
        --code to proccess @Sid, @Name, @Surname
        print 'The record with the Sid=' + CAST(@Sid AS varchar(50)) +
        ' has the name ' + CONVERT(varchar(50), @Name) +
        ' and the surname ' + CONVERT(varchar(50), @Surname)
        FETCH CursorStudents
        INTO @Sid, @Name, @Surname
END
CLOSE CursorStudents
DEALLOCATE CursorStudents
```



```
The record with the Sid=1 has the name Mihnea and the surname Dan
The record with the Sid=2 has the name Mailat and the surname Mihaela
The record with the Sid=3 has the name Roman and the surname Paul
The record with the Sid=4 has the name Romaniuc and the surname Loredana
The record with the Sid=5 has the name Cristea and the surname Mihai
The record with the Sid=6 has the name Pitic and the surname Mirela

Completion time: 2021-10-14T15:47:44.5923164+03:00
```

Query executed successfully. | DESKTOP-VB4PGBM\SQLEXPRESS ... | DESKTOP-VB4PGBM\A (53) | Seminar4 | 00:00:00 | 0 rows

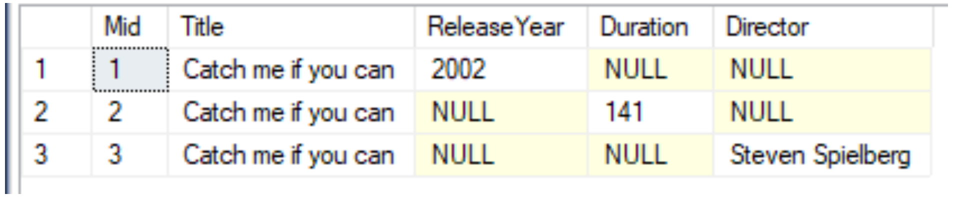
The MERGE statement

- Is a statement in which a source table is compared with a target table: INSERT, UPDATE, DELETE statements can be executed based on the result of the comparison
- INSERT / UPDATE / DELETE operations can be executed on the target table based on the result of a join with the source table.

```
MERGE TargetTable AS Target
USING SourceTable AS Source
ON (Search terms)
WHEN MATCHED THEN
            UPDATE SET or DELETE
WHEN NOT MATCHED [BY TARGET] THEN
            INSERT
WHEN NOT MATCHED [BY SOURCE] THEN
            UPDATE SET or DELETE
```

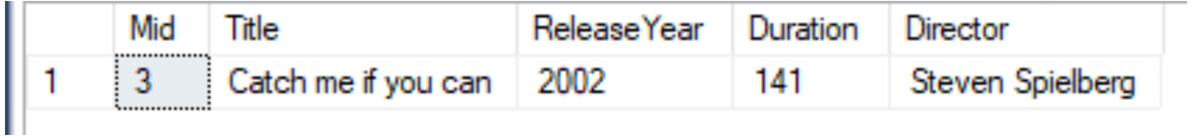
The MERGE statement

Example: The Movie table:



	Mid	Title	ReleaseYear	Duration	Director
1	1	Catch me if you can	2002	NULL	NULL
2	2	Catch me if you can	NULL	141	NULL
3	3	Catch me if you can	NULL	NULL	Steven Spielberg

```
MERGE Movie
USING
	(SELECT MAX(Mid) Mid, Title, MAX(ReleaseYear) ReleaseYear, MAX(Duration) Duration,
	MAX(Director) Director
	FROM Movie
	GROUP BY Title) MergeData ON Movie.Mid = MergeData.Mid
WHEN MATCHED THEN
	UPDATE SET Movie.Title = MergeData.Title,
	Movie.ReleaseYear = MergeData.ReleaseYear,
	Movie.Duration = MergeData.Duration,
	Movie.Director = MergeData.Director
WHEN NOT MATCHED BY SOURCE THEN DELETE;
```



	Mid	Title	ReleaseYear	Duration	Director
1	3	Catch me if you can	2002	141	Steven Spielberg