Pivot and Apply Operators, UDF, EXCEPT / INTERSECT

ISM 6218

Due on October 1st

The Avengers Team

"We will avenge every problem on our way"

Aitemir Yeskenov (Team Lead)

Nagarjuna Kanneganti

Sai Suraj Argula

Vinay Kumar Reddy Baradi

Table of Contents

Business Process Supported

We have used the Employee Database which we created for exam 1. It consists of details of employees from different states in USA. It shares details such as **EmployeeID**, **EmployeeCity**, **EmployeeFirstsame**, **EmployeeLastNameState**, **ZipCode**.

Requirements Described

Part 1:

1. Create a UDF scalar

2. Create a UDF Table Valued

3. Call function from select statement

- Must have at least 1 parameter.

4. Create a SPROC that Calls UDF, Execute SPROC

- Must have at least 2 parameters.

5. Produce Select statement result using Apply operator

- Must have at least 1 UDF and 1 Table.
- Must demonstrate 1 outer and 1 cross.
- Compare to inner, left/right, full join

6. Demonstrate use of Intersect and Except Operators

Part 2:

Create a pivot table from a current table.

Unpivot the table to a new table.

Must support a user story and reporting requirement.

Must Pivot and Unpivot both ways:

- 1. Using Cross Join and Apply
- 2. Using Pivot and Unpivot

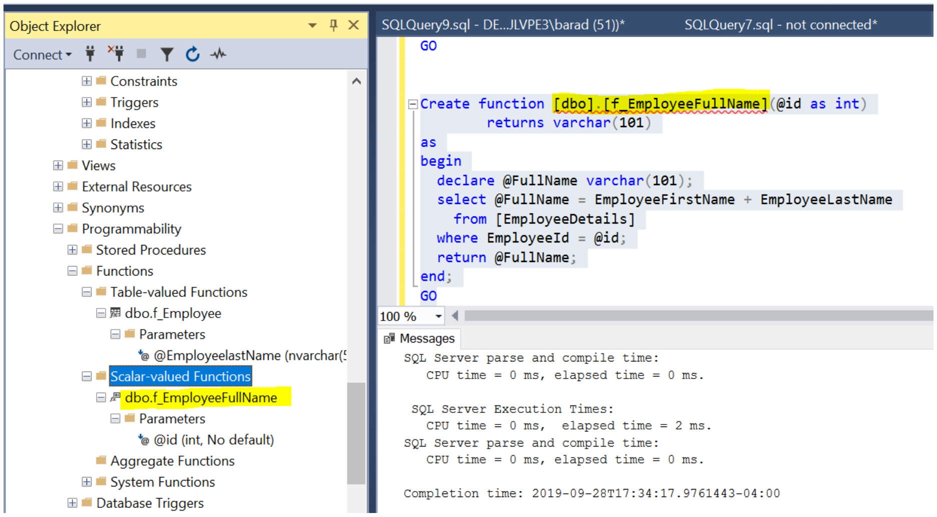
1. UDF SCALAR AND TABLE VALUED

SCALAR VALUED FUNCTIONS

I have used the EmployeeDetails database for the demonstration of Scalar and Table valued User defined functions.

A scalar UDF returns one, and only one value from the function. You can pass in parameters, then return a result.

The purpose of this function **[dbo].[f_EmployeeFullName]** is to take the Employee ID (the Primary Key) and look up the name in the EmployeeDetails table. It then concatenates the first and last names for us and returns the full name.

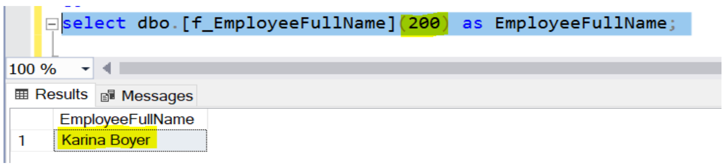


```
GO


Create function [dbo].[f_EmployeeFullName](@id as int)
        returns varchar(101)
as
begin
  declare @FullName varchar(101);
  select @FullName = EmployeeFirstName + EmployeeLastName
    from [EmployeeDetails]
  where EmployeeId = @id;
  return @FullName;
end;
GO
```

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 2 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

Completion time: 2019-09-28T17:34:17.9761443-04:00
```

I began with a create function command, followed by the name of the function. Next comes the return type, in this case I'm returning a varchar. Function will then be enclosed in a begin…end construct. I've declared a variable to hold the return value (@FullName), and then run a simple select statement to get the name, concatenate it, and store it in our variable. Finally, I use the return command to return the value back to the caller.

I passed in a Employee id(200) in to the function **[dbo].[f_EmployeeFullName** and it returned the full name of that person as shown below.

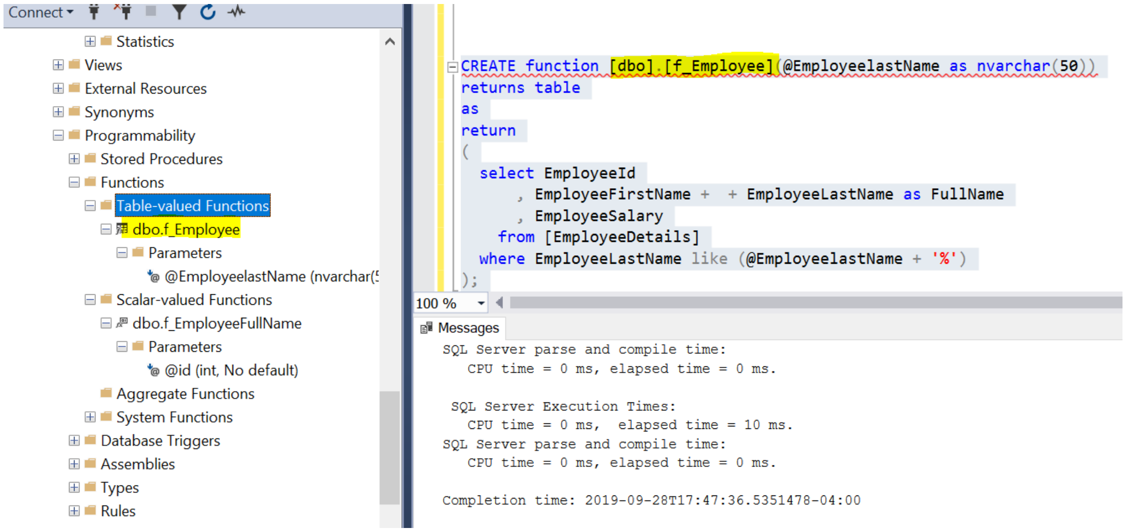


```
select dbo.[f_EmployeeFullName](200) as EmployeeFullName;
```

	EmployeeFullName
1	Karina Boyer

TABLE VALUED FUNCTIONS

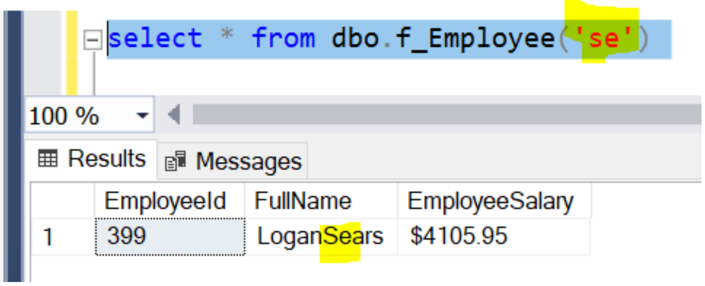
I have created table valued function **[dbo].[f_Employee**].i can use one select statement inside a function. The return type of the function is declared as table, which flags this as a table valued UDF. Here I pass in a varchar string, and use it as part of the where clause inside the UDF.



```
CREATE function [dbo].[f_Employee](@EmployeelastName as nvarchar(50))
returns table
as
return
(
  select EmployeeId
       , EmployeeFirstName +  + EmployeeLastName as FullName
       , EmployeeSalary
    from [EmployeeDetails]
  where EmployeeLastName like (@EmployeelastName + '%')
);
```

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 10 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

Completion time: 2019-09-28T17:47:36.5351478-04:00
```

Here’s an example of using it,. We are getting the FullName of the Employee whose LastName starts with ‘Se’ from EmployeeDetails Table when I passed the ‘Se’ in table valued function **[dbo].[f_Employee**].

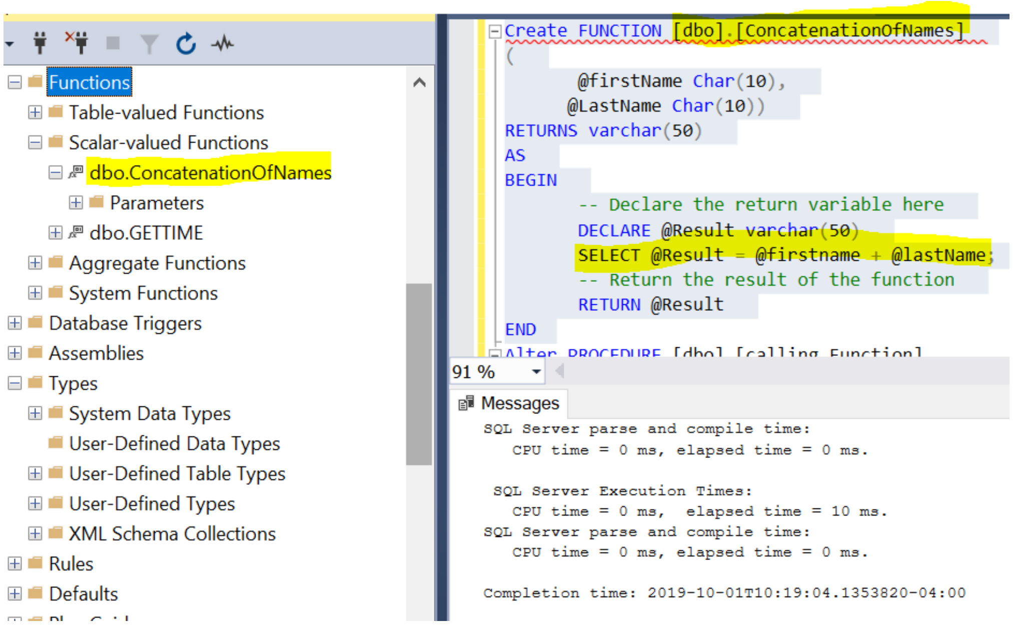


```
select * from dbo.f_Employee('se')
```

	EmployeeId	FullName	EmployeeSalary
1	399	LoganSears	$4105.95

4. Create a SPROC that Calls UDF, Execute SPROC

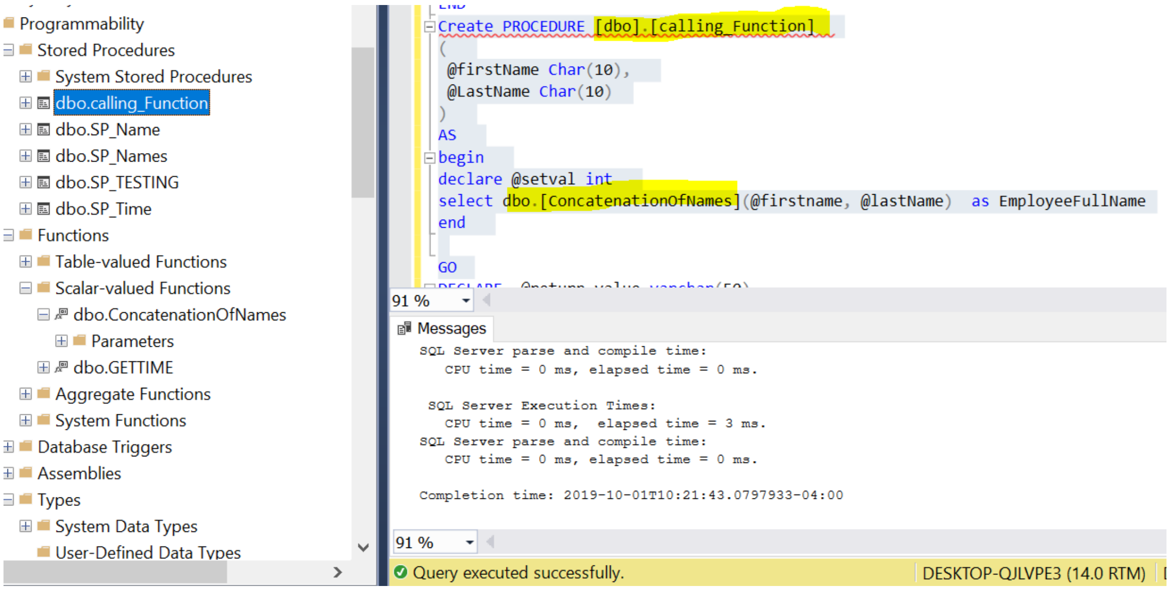
Now am demonstrating how to call a function from a stored procedure. First, I have written a scalar function named **dbo.ConcatenationOfNames** with the two parameters number1 and number2 returning one parameter named result. Both parameters have the same type, char(10). The function looks as in the following:



```
Create FUNCTION [dbo].[ConcatenationOfNames]
(
        @firstName Char(10),
       @LastName Char(10))
RETURNS varchar(50)
AS
BEGIN
        -- Declare the return variable here
        DECLARE @Result varchar(50)
        SELECT @Result = @firstname + @lastName;
        -- Return the result of the function
        RETURN @Result
END
Alter PROCEDURE [dbo].[calling_Function]
```

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 10 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

Completion time: 2019-10-01T10:19:04.1353820-04:00
```

Now I want to call this from a stored procedure. So created a stored procedure [dbo].[callingFunction] which calls a function named dbo.ConcatenationOfNames as shown in below screenshot.

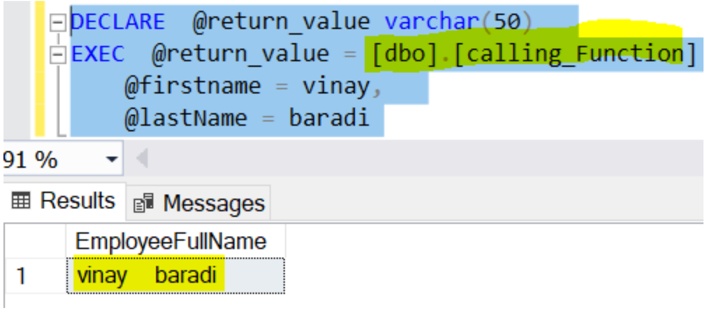


```
Create PROCEDURE [dbo].[calling_Function]
(
 @firstName Char(10),
 @LastName Char(10)
)
AS
begin
declare @setval int
select dbo.[ConcatenationOfNames](@firstname, @lastName)  as EmployeeFullName
end

GO
DECLARE  @return_value varchar(50)
```

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 3 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

Completion time: 2019-10-01T10:21:43.0797933-04:00
```

✓ Query executed successfully.

Now, we executed the procedure with duplicate values(‘vinay’,’baradi’) to check how to call a function from a procedure we created. So finally, we got the desired output ‘vinay baradi’ as shown in below screenshot.



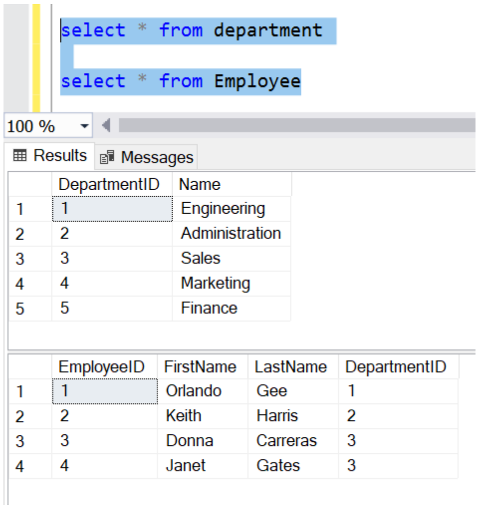
```
DECLARE  @return_value varchar(50)
EXEC  @return_value = [dbo].[calling_Function]
      @firstname = vinay,
      @lastName = baradi
```

```
   EmployeeFullName
1  vinay    baradi
```

5. Produce Select statement result using Apply operator

The need of APPLY arises when we have a table-valued expression on the right part and in some cases the use of the APPLY operator boosts performance of your query also. In the below queries we will demonstrate the inner join, Left, Right, Full joins and then Cross and outer Apply operators.

To demonstrate we have created two tables ‘Department’ and ‘Employee’ as listed below.



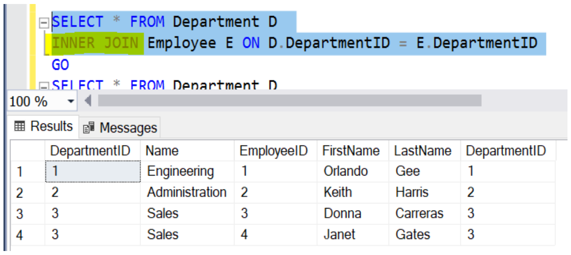
```
select * from department

select * from Employee
```

	DepartmentID	Name
1	1	Engineering
2	2	Administration
3	3	Sales
4	4	Marketing
5	5	Finance

	EmployeeID	FirstName	LastName	DepartmentID
1	1	Orlando	Gee	1
2	2	Keith	Harris	2
3	3	Donna	Carreras	3
4	4	Janet	Gates	3

INNER JOIN

This query simply joins the Department table with the Employee table and all matching records are produced.

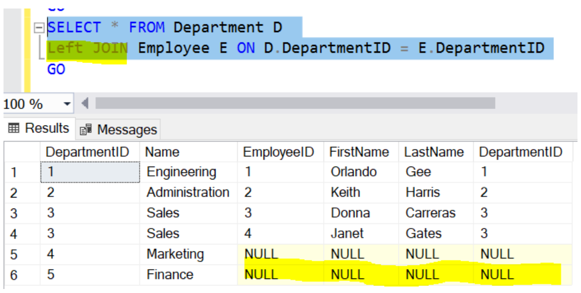


```
SELECT * FROM Department D
INNER JOIN Employee E ON D.DepartmentID = E.DepartmentID
GO
SELECT * FROM Department D
```

	DepartmentID	Name	EmployeeID	FirstName	LastName	DepartmentID
1	1	Engineering	1	Orlando	Gee	1
2	2	Administration	2	Keith	Harris	2
3	3	Sales	3	Donna	Carreras	3
4	3	Sales	4	Janet	Gates	3

LEFT JOIN

Query simply uses a LEFT OUTER JOIN between the Department table and the Employee table. As expected, the query returns all rows from Department table, even for those rows for which there is no match in the Employee table.

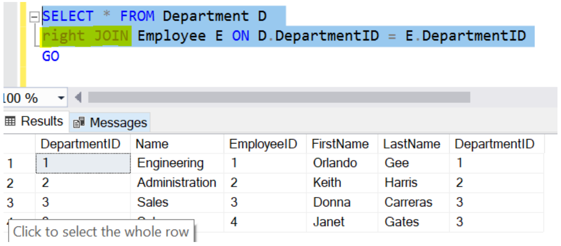


```
SELECT * FROM Department D
Left JOIN Employee E ON D.DepartmentID = E.DepartmentID
GO
```

	DepartmentID	Name	EmployeeID	FirstName	LastName	DepartmentID
1	1	Engineering	1	Orlando	Gee	1
2	2	Administration	2	Keith	Harris	2
3	3	Sales	3	Donna	Carreras	3
4	3	Sales	4	Janet	Gates	3
5	4	Marketing	NULL	NULL	NULL	NULL
6	5	Finance	NULL	NULL	NULL	NULL

RIGHT JOIN:

query simply uses a LEFT OUTER JOIN between the Department table and the Employee table. As expected the query returns all rows from Employee table, even for those rows for which there is no match in the Department table.

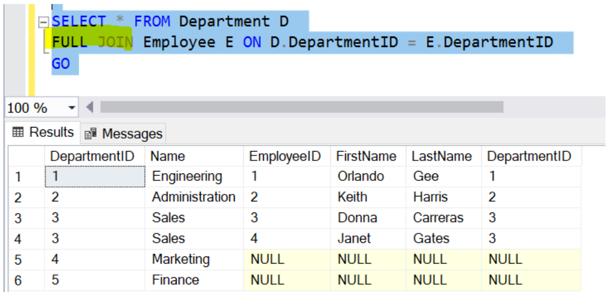


```
SELECT * FROM Department D
right JOIN Employee E ON D.DepartmentID = E.DepartmentID
GO
```

	DepartmentID	Name	EmployeeID	FirstName	LastName	DepartmentID
1	1	Engineering	1	Orlando	Gee	1
2	2	Administration	2	Keith	Harris	2
3	3	Sales	3	Donna	Carreras	3
			4	Janet	Gates	3

FULL JOIN

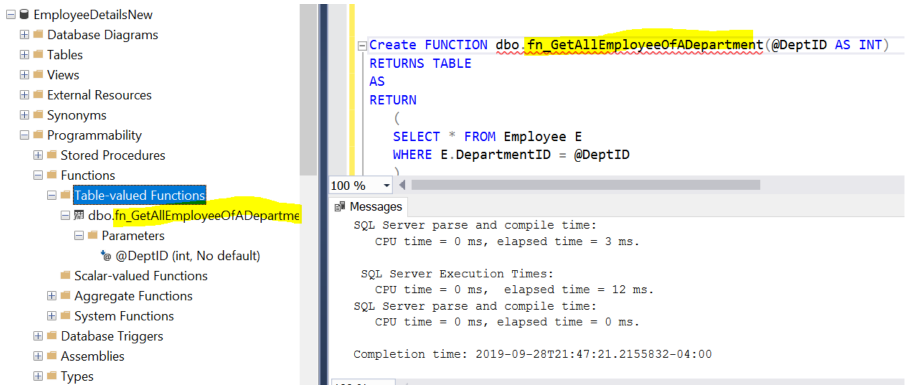
The **FULL JOIN** combines the results of both left and right outer **joins** and returns all (matched or unmatched) rows from the tables on both sides of the **join** clause as seen below.



```
SELECT * FROM Department D
FULL JOIN Employee E ON D.DepartmentID = E.DepartmentID
GO
```

	DepartmentID	Name	EmployeeID	FirstName	LastName	DepartmentID
1	1	Engineering	1	Orlando	Gee	1
2	2	Administration	2	Keith	Harris	2
3	3	Sales	3	Donna	Carreras	3
4	3	Sales	4	Janet	Gates	3
5	4	Marketing	NULL	NULL	NULL	NULL
6	5	Finance	NULL	NULL	NULL	NULL

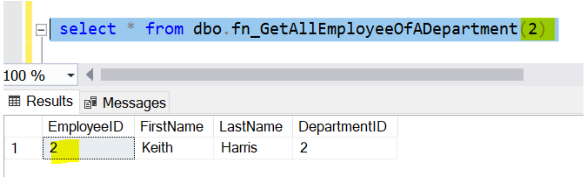
Now I am creating a table-valued function **dbo.fn_GetAllEmployeeOfADepartment** which accepts DepartmentID as its parameter and returns all the employees who belong to this department



```
Create FUNCTION dbo.fn_GetAllEmployeeOfADepartment(@DeptID AS INT)
RETURNS TABLE
AS
RETURN
    (
    SELECT * FROM Employee E
    WHERE E.DepartmentID = @DeptID
    )
```

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 3 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 12 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

Completion time: 2019-09-28T21:47:21.2155832-04:00
```

For example, we have passed the departmentId ‘2’ has the parameter in to the **dbo.fn_GetAllEmployeeOfADepartment** which returns the table with EmployeeID, FirstName, LastName, DepartmentId as shown below.

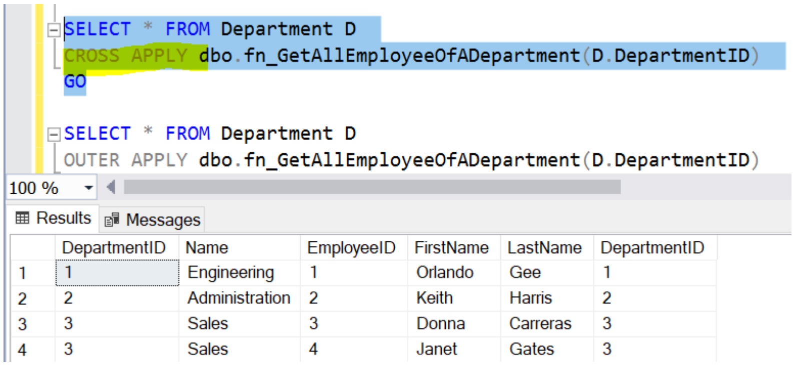


```
select * from dbo.fn_GetAllEmployeeOfADepartment(2)
```

	EmployeeID	FirstName	LastName	DepartmentID
1	2	Keith	Harris	2

CROSS APPLY

Next query selects data from the Department table and uses a CROSS APPLY to join with the function we created. It passes the DepartmentID for each row from the Department table and evaluates the function for each row as shown below. The CROSS APPLY operator returns only those rows from the left table expression (in its final output) if it matches with the right table expression.

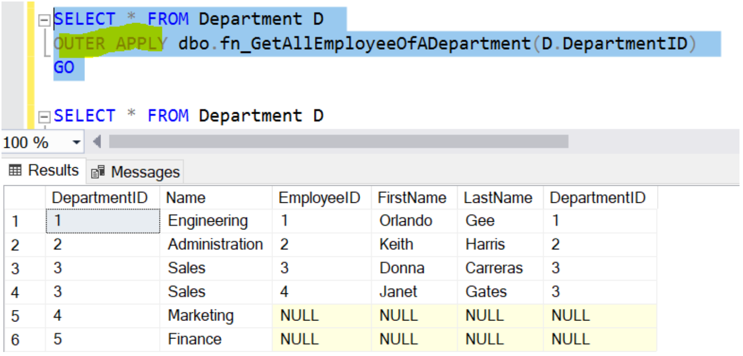


```
SELECT * FROM Department D
CROSS APPLY dbo.fn_GetAllEmployeeOfADepartment(D.DepartmentID)
GO

SELECT * FROM Department D
OUTER APPLY dbo.fn_GetAllEmployeeOfADepartment(D.DepartmentID)
```

	DepartmentID	Name	EmployeeID	FirstName	LastName	DepartmentID
1	1	Engineering	1	Orlando	Gee	1
2	2	Administration	2	Keith	Harris	2
3	3	Sales	3	Donna	Carreras	3
4	3	Sales	4	Janet	Gates	3

OUTER APPLY

This query uses the OUTER APPLY in place of the CROSS APPLY and hence unlike the CROSS APPLY which returned only correlated data, the OUTER APPLY returns non-correlated data as well, placing NULLs into the missing columns.



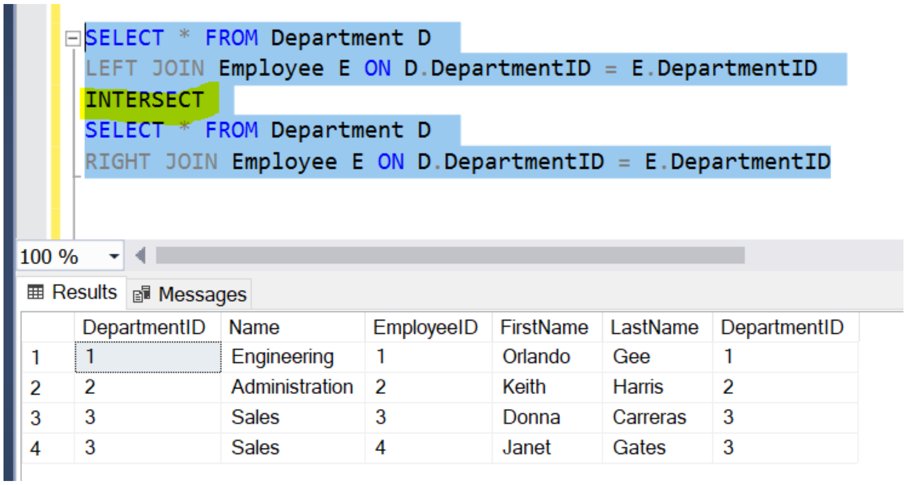
```
SELECT * FROM Department D
OUTER APPLY dbo.fn_GetAllEmployeeOfADepartment(D.DepartmentID)
GO

SELECT * FROM Department D
```

	DepartmentID	Name	EmployeeID	FirstName	LastName	DepartmentID
1	1	Engineering	1	Orlando	Gee	1
2	2	Administration	2	Keith	Harris	2
3	3	Sales	3	Donna	Carreras	3
4	3	Sales	4	Janet	Gates	3
5	4	Marketing	NULL	NULL	NULL	NULL
6	5	Finance	NULL	NULL	NULL	NULL

If you replace the CROSS/OUTER APPLY in the above queries with an INNER JOIN/LEFT OUTER JOIN, specifying the ON clause and run the query, you will get the error "The multi-part identifier "D.DepartmentID" could not be bound.". This is because with JOINs the execution context of the outer query is different from the execution context of the function and you cannot bind a value/variable from the outer query to the function as a parameter. Hence the APPLY operator is required for such queries. So in summary the APPLY operator is required when you have to use a table-valued function in the query.

6. Demonstrate use of Intersect and Except Operators

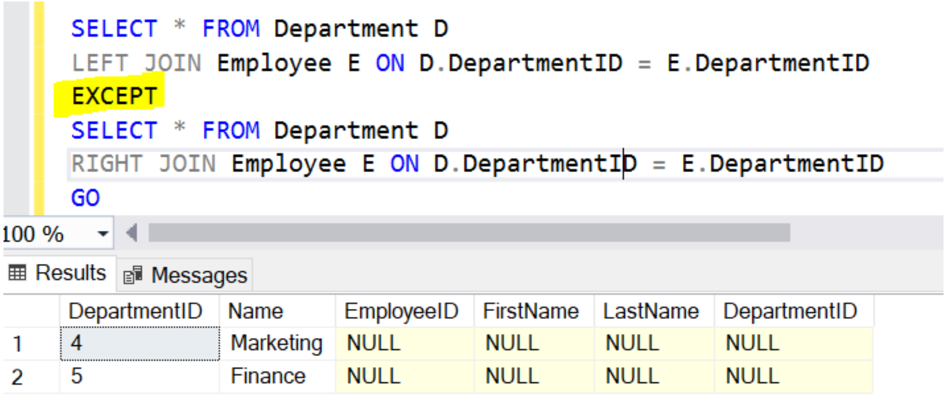
INTERSECT returns rows that are in **common** between two Department and Employee Table; This query is useful when you want to find results that are in common between two queries.



```
SELECT * FROM Department D
LEFT JOIN Employee E ON D.DepartmentID = E.DepartmentID
INTERSECT
SELECT * FROM Department D
RIGHT JOIN Employee E ON D.DepartmentID = E.DepartmentID
```

	DepartmentID	Name	EmployeeID	FirstName	LastName	DepartmentID
1	1	Engineering	1	Orlando	Gee	1
2	2	Administration	2	Keith	Harris	2
3	3	Sales	3	Donna	Carreras	3
4	3	Sales	4	Janet	Gates	3

We use **Except Operator** to return only rows found in the left query. It returns **unique** rows from the left query that aren’t in the right query’s results. This query is useful when you’re looking to find rows that are in one set but not another



```sql
SELECT * FROM Department D
LEFT JOIN Employee E ON D.DepartmentID = E.DepartmentID
EXCEPT
SELECT * FROM Department D
RIGHT JOIN Employee E ON D.DepartmentID = E.DepartmentID
GO
```

	DepartmentID	Name	EmployeeID	FirstName	LastName	DepartmentID
1	4	Marketing	NULL	NULL	NULL	NULL
2	5	Finance	NULL	NULL	NULL	NULL

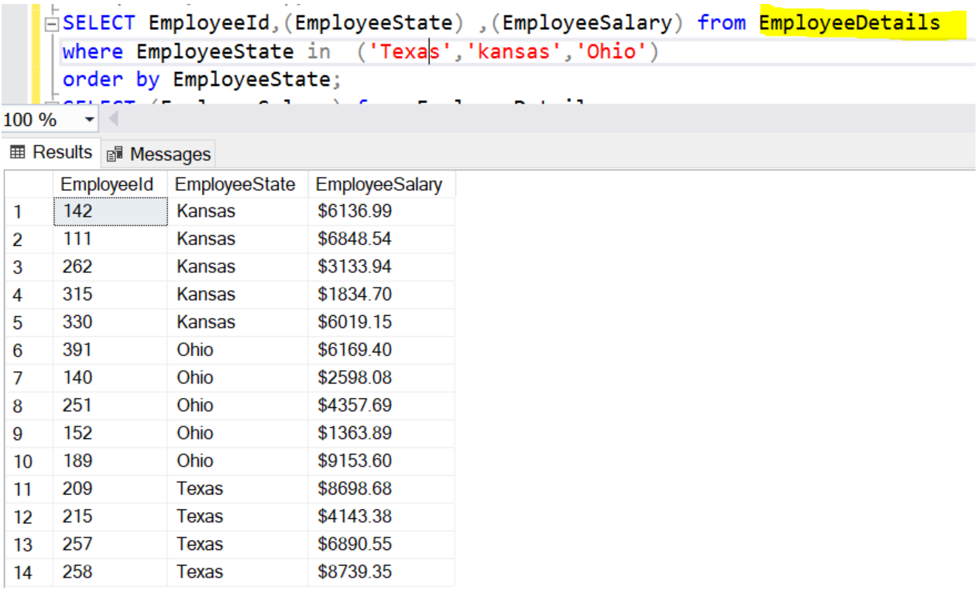
Part 2

Pivot and Unpivot

Using PIVOT Operator:

Here I am using the PIVOT operator to get the maximum salary of the employee in each state.

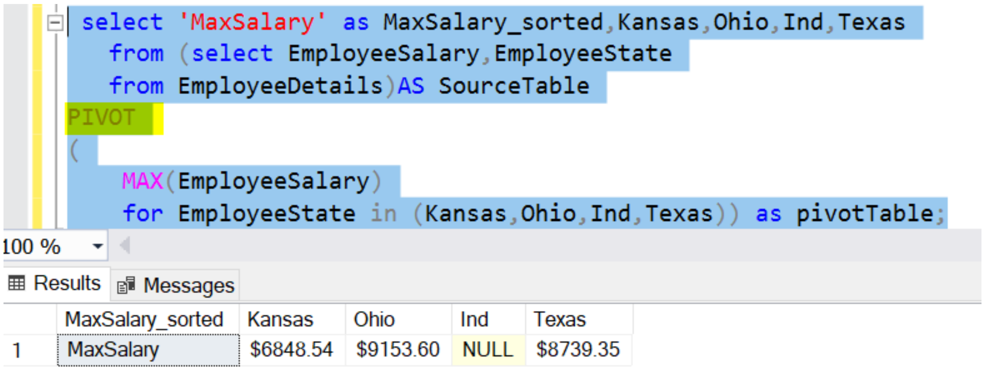
I have used the EmployeeDetails table which return the Employee Salary, EmployeeState and EmployeeId as output.



```sql
SELECT EmployeeId,(EmployeeState) ,(EmployeeSalary) from EmployeeDetails
where EmployeeState in  ('Texas','kansas','Ohio')
order by EmployeeState;
```

	EmployeeId	EmployeeState	EmployeeSalary
1	142	Kansas	$6136.99
2	111	Kansas	$6848.54
3	262	Kansas	$3133.94
4	315	Kansas	$1834.70
5	330	Kansas	$6019.15
6	391	Ohio	$6169.40
7	140	Ohio	$2598.08
8	251	Ohio	$4357.69
9	152	Ohio	$1363.89
10	189	Ohio	$9153.60
11	209	Texas	$8698.68
12	215	Texas	$4143.38
13	257	Texas	$6890.55
14	258	Texas	$8739.35

We have applied the Pivot operator as shown below to get the **maximum salary** of the employee in each state with State as columns and maximum salary as the rows in result.



```
select 'MaxSalary' as MaxSalary_sorted,Kansas,Ohio,Ind,Texas
  from (select EmployeeSalary,EmployeeState
  from EmployeeDetails)AS SourceTable
PIVOT
(
   MAX(EmployeeSalary)
   for EmployeeState in (Kansas,Ohio,Ind,Texas)) as pivotTable;
```

	MaxSalary_sorted	Kansas	Ohio	Ind	Texas
1	MaxSalary	$6848.54	$9153.60	NULL	$8739.35

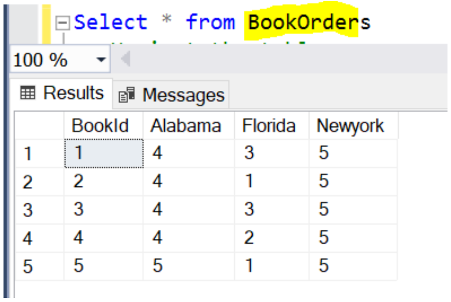
Unpivot

1. Using Unpivot operator

For Unpivot operator explanation, I have created the BookOrders Table. Table consists of the number of book orders from Alabama, Florida, Newyork states ,For Example, the BookId ‘1’ has 4 orders from Alabama followed by 3,5 from Florida, Newyork respectively.

Now, I will be using the unpivot operator to get the number of orders of each BookId for the states.

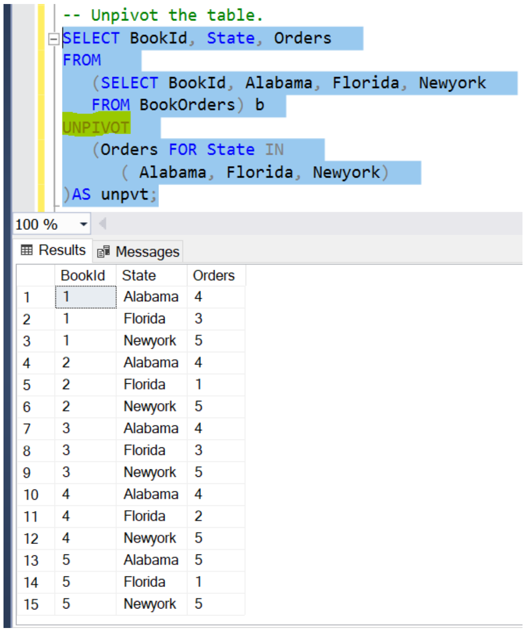
We used the Unpivot operator to get the desired result. UNPIVOT carries out almost the reverse operation of PIVOT, by rotating columns into rows.



```
Select * from BookOrders
```

	BookId	Alabama	Florida	Newyork
1	1	4	3	5
2	2	4	1	5
3	3	4	3	5
4	4	4	2	5
5	5	5	1	5

The column that will contain the column values that are rotating (Alabama, Florida,Newyork) will be called State, and the column that will hold the values that currently exist under the columns being rotated will be called Orders

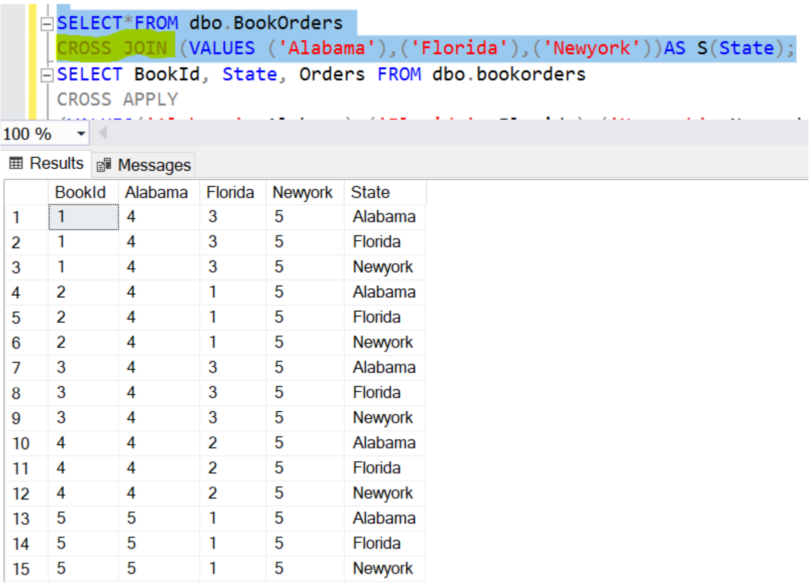


```
-- Unpivot the table.
SELECT BookId, State, Orders
FROM
   (SELECT BookId, Alabama, Florida, Newyork
   FROM BookOrders) b
UNPIVOT
   (Orders FOR State IN
      ( Alabama, Florida, Newyork)
)AS unpvt;
```

	BookId	State	Orders
1	1	Alabama	4
2	1	Florida	3
3	1	Newyork	5
4	2	Alabama	4
5	2	Florida	1
6	2	Newyork	5
7	3	Alabama	4
8	3	Florida	3
9	3	Newyork	5
10	4	Alabama	4
11	4	Florida	2
12	4	Newyork	5
13	5	Alabama	5
14	5	Florida	1
15	5	Newyork	5

2. Using Cross Join and Apply

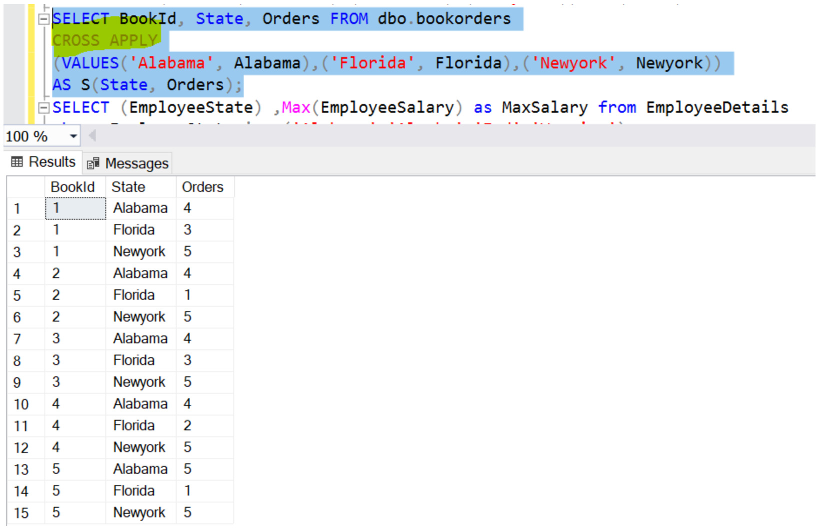
In this we have used the cross join and apply instead of the Unpivot operator to get the number of orders of each BookId in Alabama, Florida and Newyork. First, we have used the CROSS JOIN followed by Cross Apply Operator.



```
SELECT*FROM dbo.BookOrders
CROSS JOIN (VALUES ('Alabama'),('Florida'),('Newyork'))AS S(State);
SELECT BookId, State, Orders FROM dbo.bookorders
CROSS APPLY
```

	BookId	Alabama	Florida	Newyork	State
1	1	4	3	5	Alabama
2	1	4	3	5	Florida
3	1	4	3	5	Newyork
4	2	4	1	5	Alabama
5	2	4	1	5	Florida
6	2	4	1	5	Newyork
7	3	4	3	5	Alabama
8	3	4	3	5	Florida
9	3	4	3	5	Newyork
10	4	4	2	5	Alabama
11	4	4	2	5	Florida
12	4	4	2	5	Newyork
13	5	5	1	5	Alabama
14	5	5	1	5	Florida
15	5	5	1	5	Newyork

Here we can see the number of orders of each BookId in Alabama, Florida and Newyork in output using the **Cross Apply** Operator. So we received the same output we received using the unpivot operator as shown in below screenshot.



```
SELECT BookId, State, Orders FROM dbo.bookorders
CROSS APPLY
(VALUES('Alabama', Alabama),('Florida', Florida),('Newyork', Newyork))
AS S(State, Orders);
SELECT (EmployeeState) ,Max(EmployeeSalary) as MaxSalary from EmployeeDetails
```

	BookId	State	Orders
1	1	Alabama	4
2	1	Florida	3
3	1	Newyork	5
4	2	Alabama	4
5	2	Florida	1
6	2	Newyork	5
7	3	Alabama	4
8	3	Florida	3
9	3	Newyork	5
10	4	Alabama	4
11	4	Florida	2
12	4	Newyork	5
13	5	Alabama	5
14	5	Florida	1
15	5	Newyork	5