Exam 2

ISM 6218

Due on October 16

Vinay Kumar Reddy Baradi

**Table of Contents**

Call 1 UDF from a SPROC\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_5

Call 1 UDF from Cursor \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 7

Call 1 UDF from CTE \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_8

Coalesce Union, Union ALL, \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_11

Coalesce, isnull, exist, having, IN, Any\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_13

Use Apply operator to join UDF with another table\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 15

Intersect and Except Operators\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_17

Create a Parameterized SQL Query\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_18

Create any query using a set of Case Expressions\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_20

Create 2 Sub queries: 1 correlated, 1 uncorrelated\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_23

**Business Process Supported**

The database created for the Exam2 is the VRB\_EXAM2 database. It consists of the various details of the employees.

**Requirements Described**

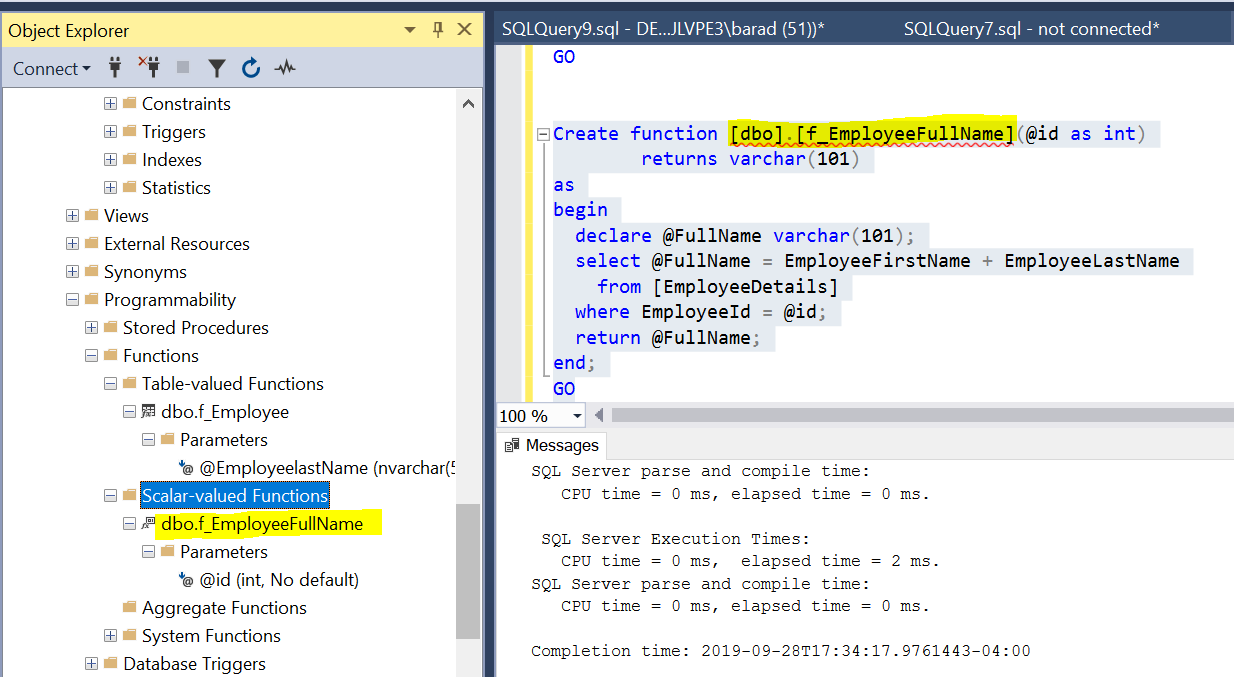
* **Create 3 User Defined Functions, using all of the following:**
  + **Union, Union ALL, Coalesce, isnull, exist, having, IN, Any**
  + **Except and Intersect operators**
  + **1 must return a table, 2 must return a single value**
  + **Call 1 UDF from a SPROC**
  + **Call 1 UDF from Cursor**
  + **Call 1 UDF from CTE**
  + **Use Apply operator to join UDF with another table**
* **Create a Parameterized SQL Query**
  + **Use Exec SQL**
  + **Use @SQL**
* **Create 2 Sub queries: 1 correlated, 1 uncorrelated**
* **Create any query using a set of Case Expressions**

**UDF WITH SPROC:**

**USER STORY:**

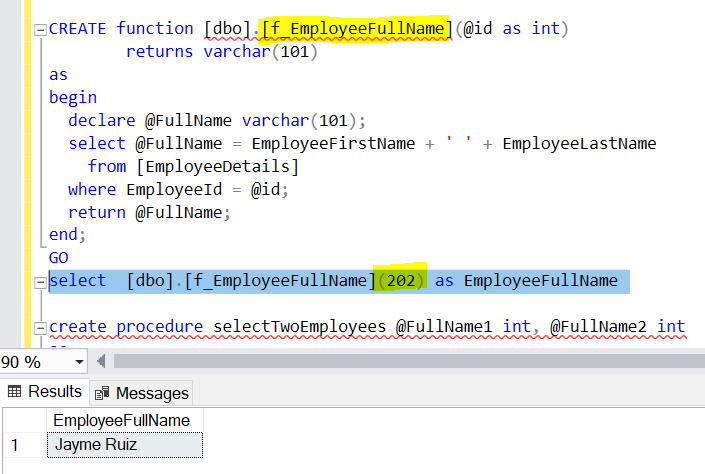
I want to get the concatenation of the firstname and the lastname of the employee as fullname when we pass the EmployeeId as parameters to the function.

I have used the EmployeeDetails database for the demonstration of 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.

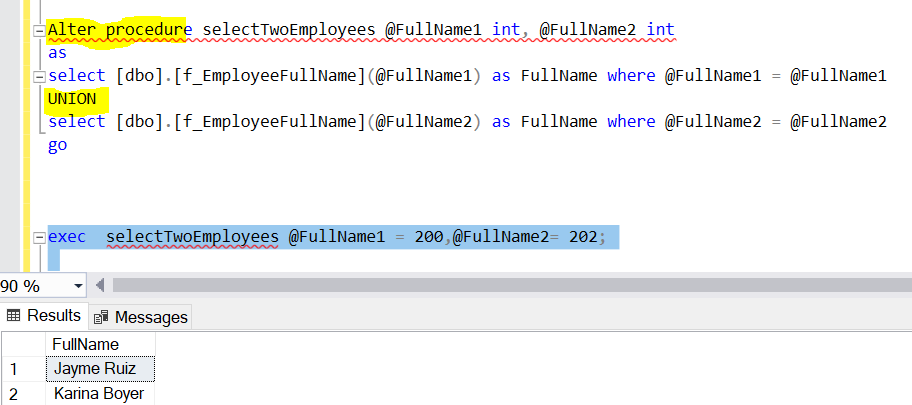


I begun with a create function command, followed by the name of the function. Next comes the return type, in this case I’m returning a varchar. Function will then be enclosed in a begin…end construct. I’ve declared a variable to hold the return value (@FullName), and then run a simple select statement 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.

 When I passed in a Employee id(202) in to the function **[dbo].[f\_EmployeeFullName** and it returned the full name of that person as shown below.



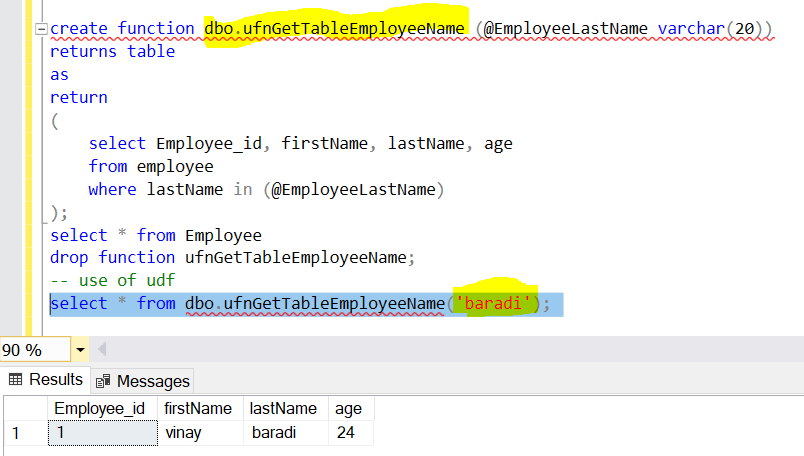
Here I have created a procedure which returns the fullname of the employeeId 200,202,



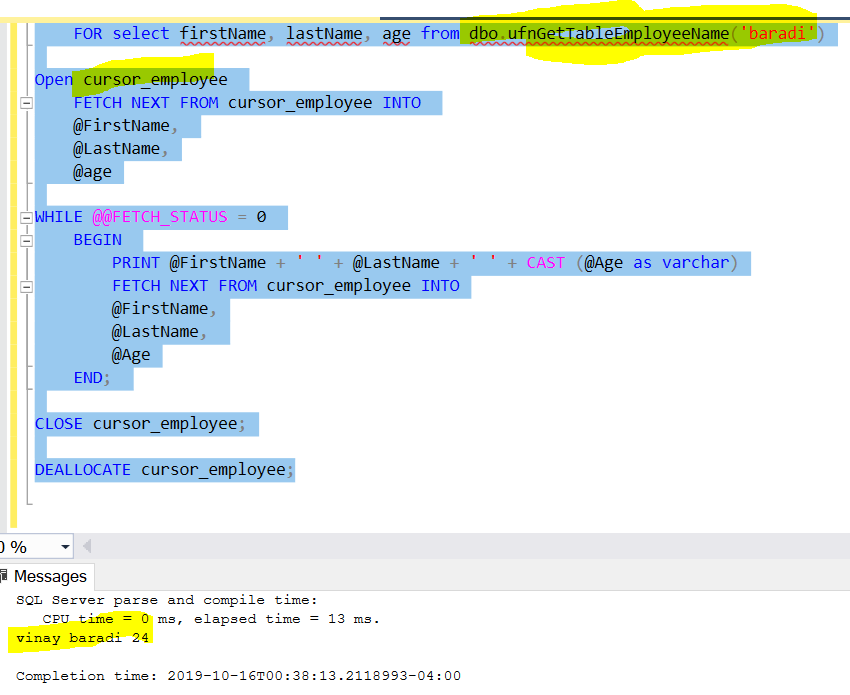
**Call UDF From Cursor :**

**USER STORY:**

We want to retrieve the age of the Employee whose last name is ‘baradi’.



Here we called the UDF from the cursor.Note that we declared the parameters based on the values of columns we want to fetch It is worth to mention that we have to open and close the cursor as well as deallocate after we achieved what we originally wanted.



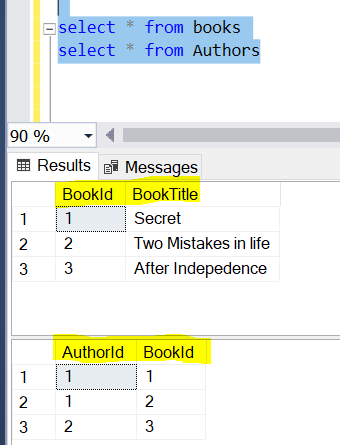
**Call UDF From CTE:**

**USER STORY:**

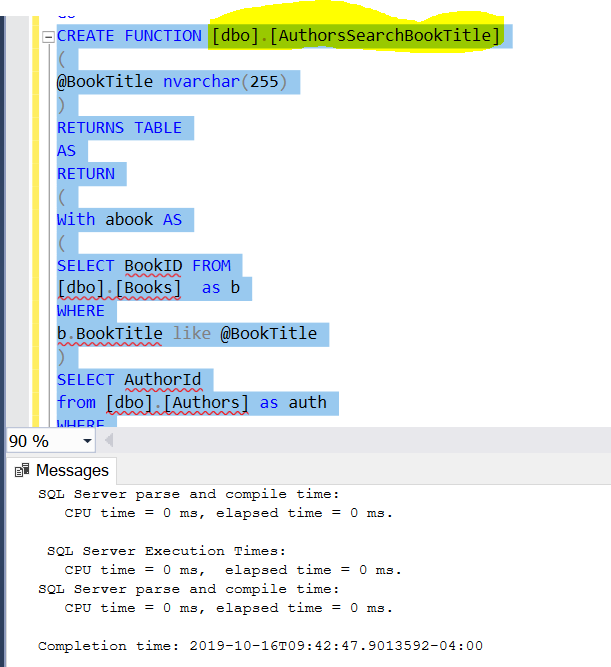
We wantretrieve the AuthorId of the particular BookTitle using the CTE.

here we have simple Book and author Tables which consists of the BookID,BookTitle

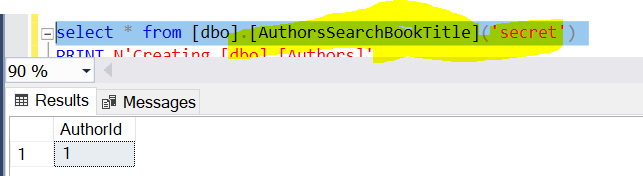
and in Author table we have AuthorId and BookId columns.



We have created a UDF table valued function with CTE abook whihch selects the BookId from Books table and AuthorId from the Authors and and return a table with the AuthorID of ht particular BookTitle we are looking for.

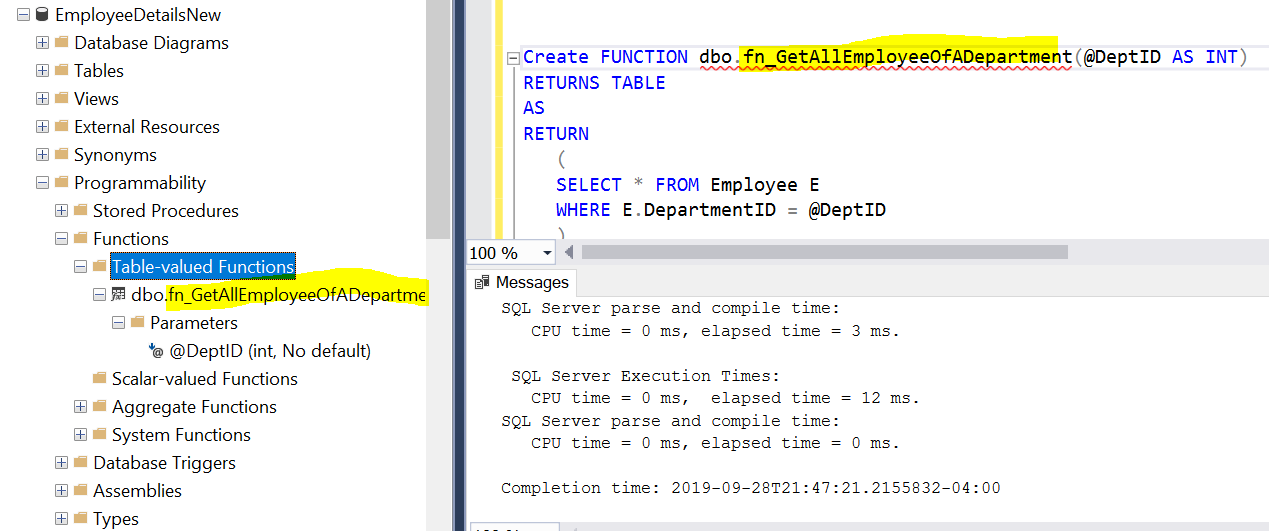


As shown below, we when we called the function with the BookTitle Secret, we received the AuthorID of the ‘Secret’ book.

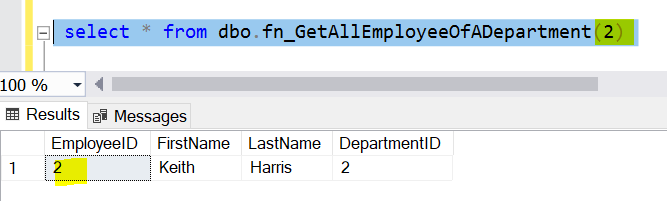


**Table Valued UDF:**

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



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.

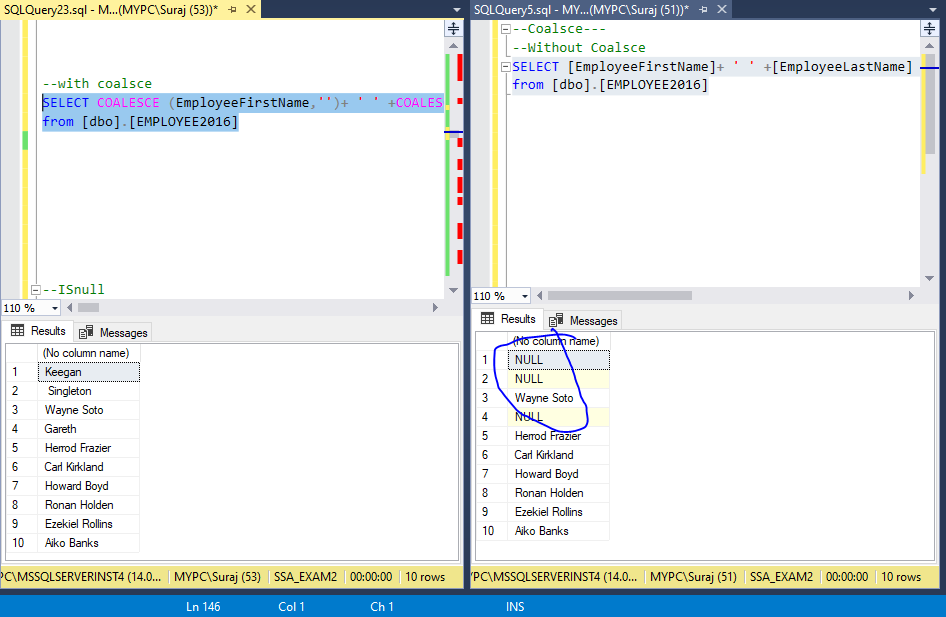


**Coalesce**

**User Story**:-

Employer wants to know the full names of all the employees to send salary slip mails.

I have created concatenation of all the Employee first names and last names, but found it return null and used coalesce to generate full names of Employees



**UNION and UNION ALL**

**User Story :-**

Employer wants to know the cities of all the employees in the Company from 2016, 2017, so that he/she can group salary slip mails to be sent.

I have created Union all of all the Employee cities from 2016,2017 employee tables, but it has duplicates to remove that I have used union. Below is the demonstration.

CREATE FUNCTION EmployeefromYears()

RETURNS TABLE

AS

RETURN

(select EmployeeCity from Employee2016

union

select EmployeeCity from EMPLOYEE2017)

select \* from Employee2016

--UNION ALL

CREATE FUNCTION EmployeeallfromYears()

RETURNS TABLE

AS

RETURN

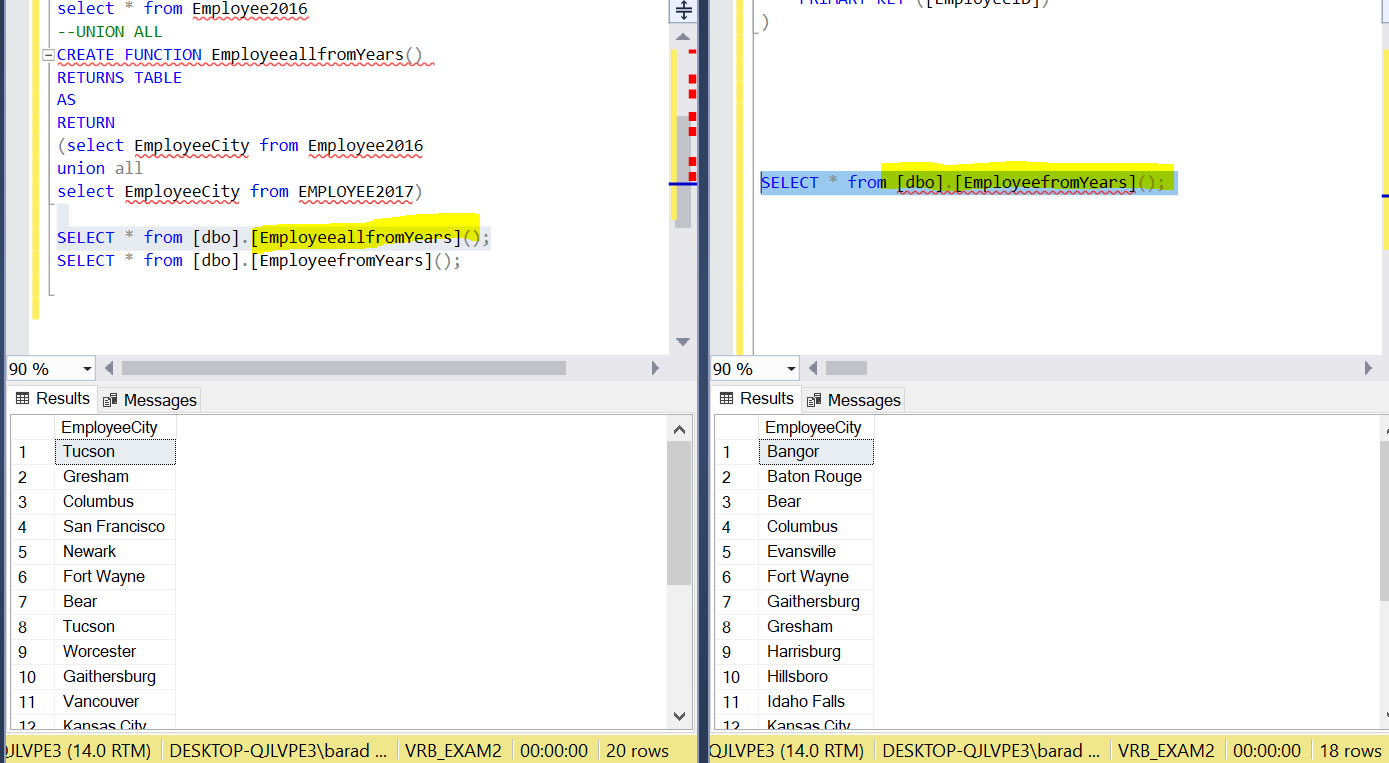
(select EmployeeCity from Employee2016

union all

select EmployeeCity from EMPLOYEE2017)

SELECT \* from [dbo].[EmployeeallfromYears]();

SELECT \* from [dbo].[EmployeefromYears]();

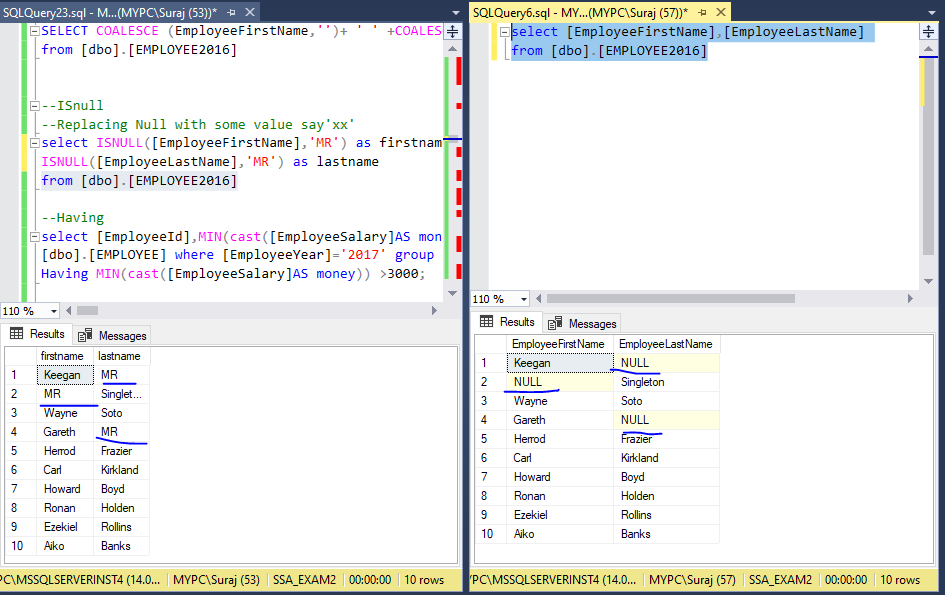


**IS NULL:-**

User Story:-

Employer wants to know if there are any Null values to employee names and wants to replace them with salutation ‘MR’.

Used ISNULL operator to replace null values with ‘MR’

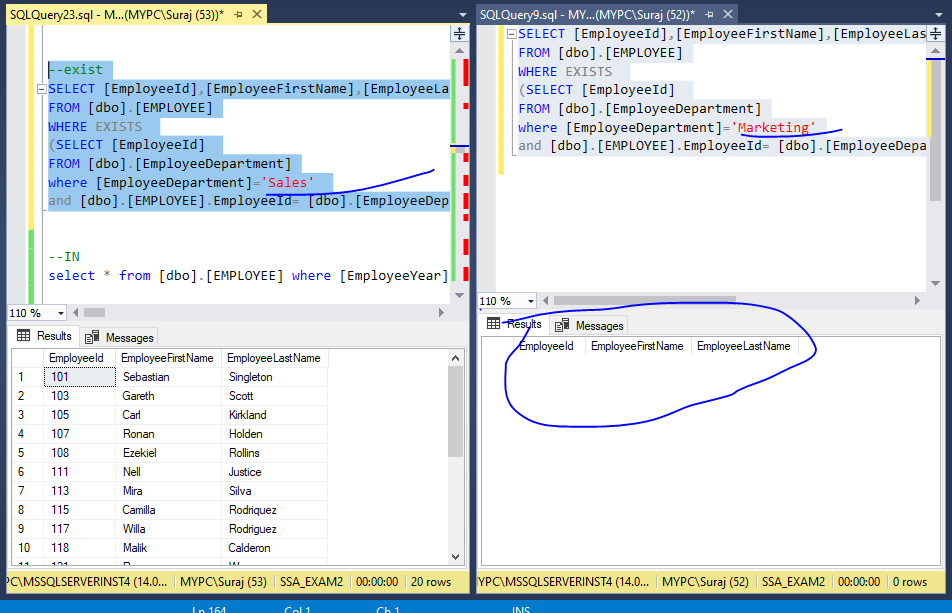


**Exists:-**

**User Story**

Employer wants to know all the Employee details of Sales and marketing people.

Used Exists operator to generate table of employee with department’s sales and marketing. Since marketing department has been discontinued and no employees present in the department. The subquery return false and main query is null.

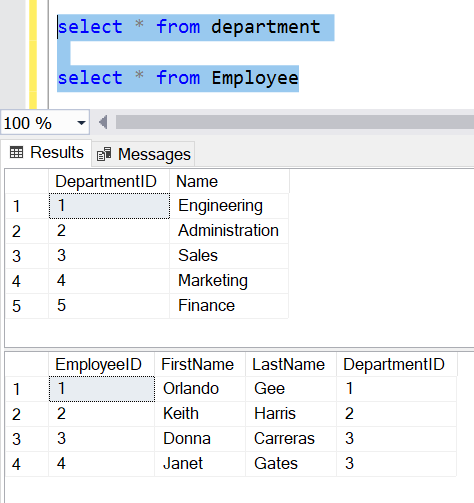


**Use Apply operator to join UDF with another table**

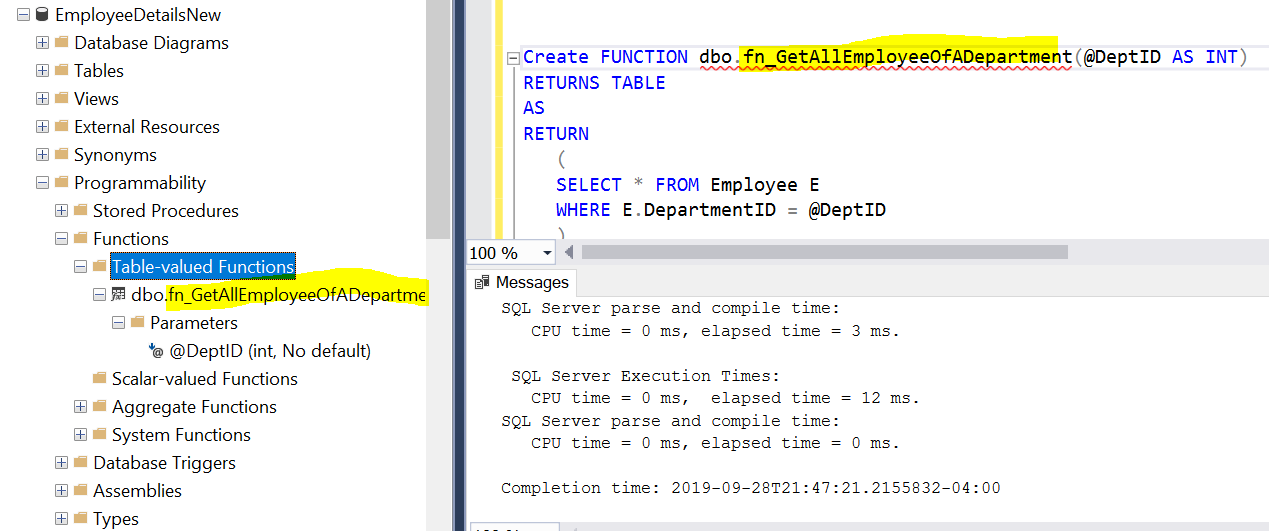
**APPLY OPERATOR:**

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](https://www.mssqltips.com/sql-server-tip-category/9/performance-tuning/) 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.

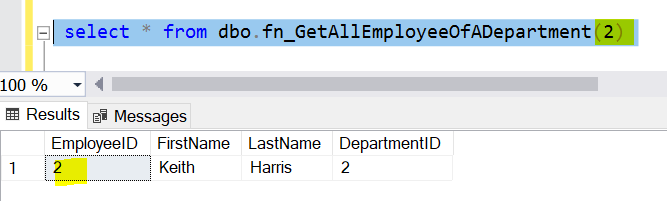
To demonstrate we have created two tables ‘Department’ and ‘Employee’ as listed below.



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

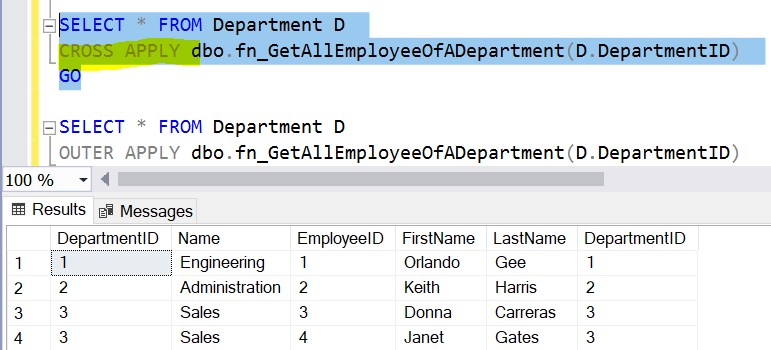


For example, we have passed the departmentId ‘2’ has the parameter in to the **dbo.fn\_GetAllEmployeeOfADepartment** which returns the table with EmployeeID, FirstName, LastName, DepartmentId as shown below.



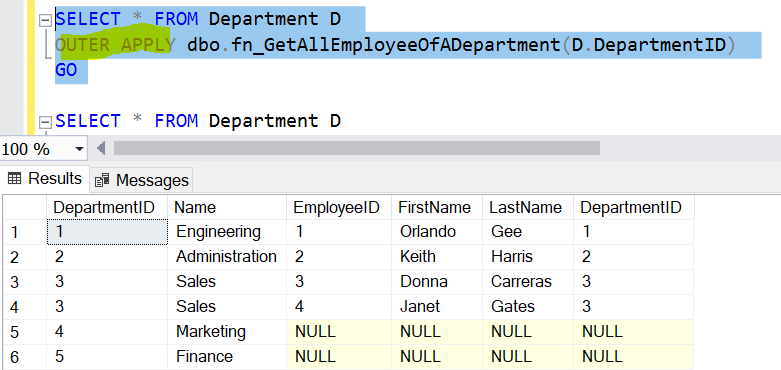
**CROSS APPLY**

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.



**OUTER APPLY**

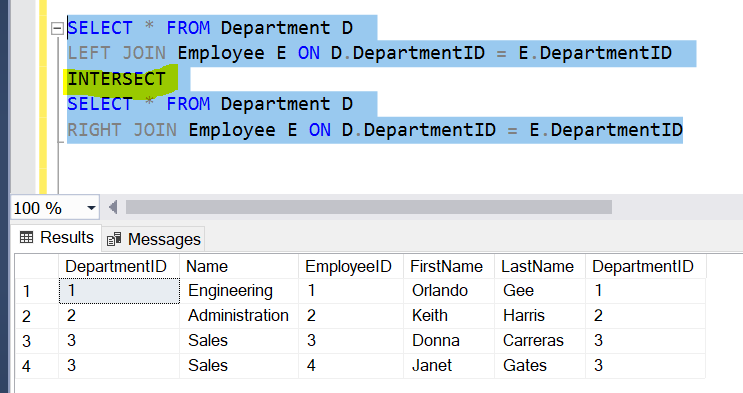
This query uses the OUTER APPLY in place of the CROSS APPLY and hence unlike the CROSS APPLY which returned only correlated data, the OUTER APPLY returns non-correlated data as well, placing NULLs into the missing columns.



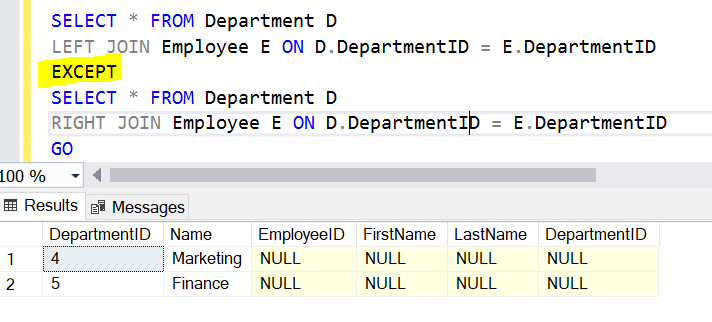
If you replace the CROSS/OUTER APPLY in the above queries with an INNER JOIN/LEFT OUTER JOIN, specifying the ON clause and run the query, you will get the error "The multi-part identifier "D.DepartmentID" could not be bound.". This is because with JOINs the execution context of the outer query is different 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.

**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.



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



**Parameterized SQL Query**

**USER STORY:**

We want to get the employee details in the Columbus city.

--Using EXEC SQL

ALTER PROC EmployeesfromCity @city nvarchar(20)

AS

BEGIN

SELECT \* FROM [dbo].[EMPLOYEE2019]

WHERE [EmployeeCity] = @city

END

set statistics time on

exec EmployeesfromCity @city ='columbus'

set statistics time off

--Using @SQL

set statistics time on

DECLARE @SQL varchar(1000)

DECLARE @city varchar(20)

SET @city ='''columbus'''

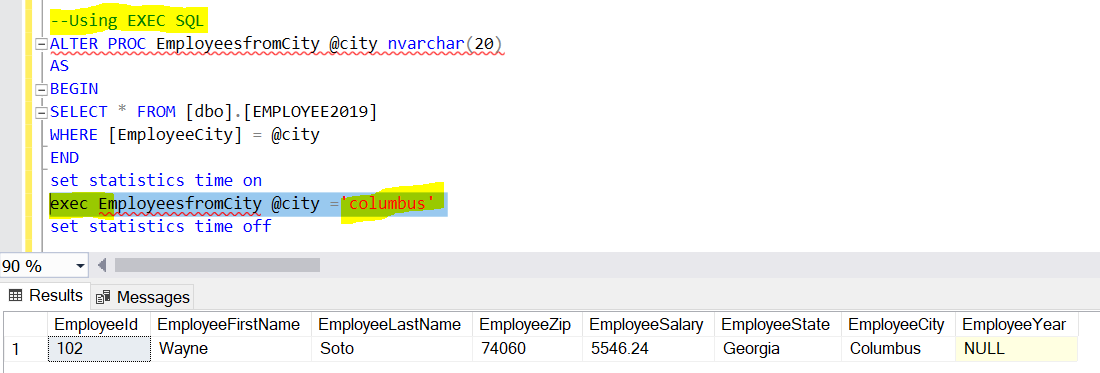
SET @SQL ='SELECT \* FROM [dbo].[EMPLOYEE2019]

WHERE [EmployeeCity] = '+ @city

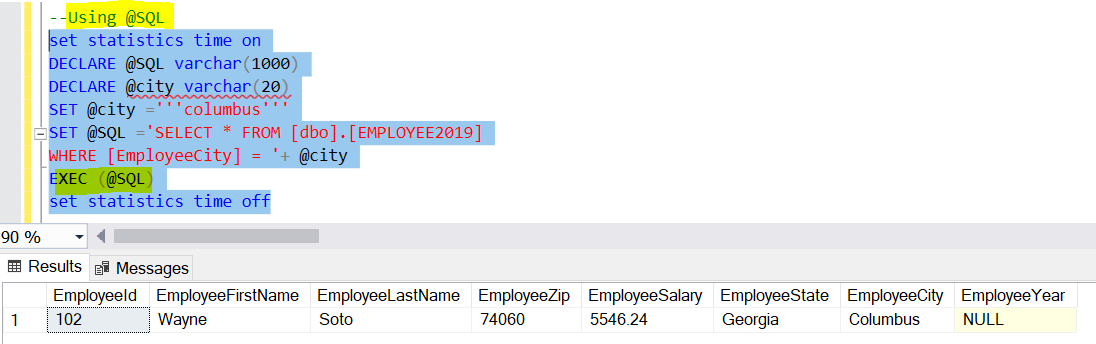
EXEC (@SQL)

set statistics time off

First, we demonstrated using the EXEC, as shown below first we created the procedure then we passed the city paramer in to the EXEC .



Here in the EXEC SQL itself, we have declared the city name.



**Create any query using a set of Case Expressions**

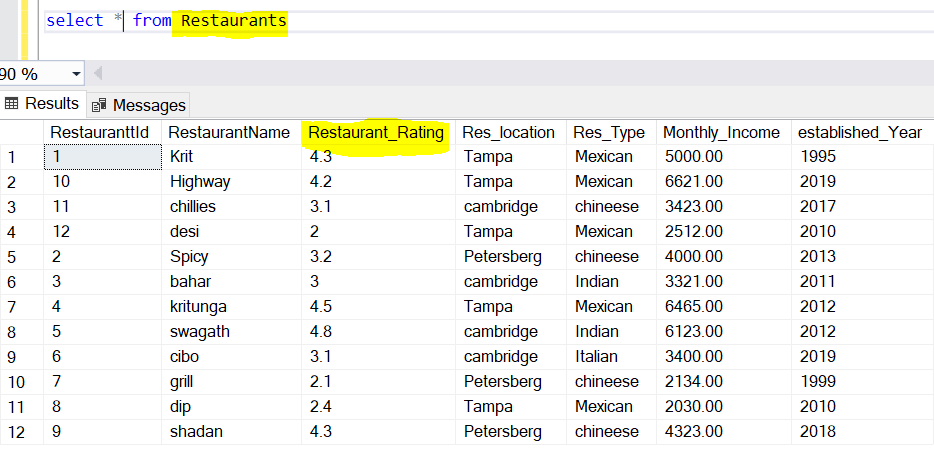
**User Story**:

We want to produce a summary table which gives information about the number of restaurants in the Tampa location having the food **Rating Category** as Great, Average, Poor using the Case Expression.

We created a simple **Restaurant** Table which consists of the columns - RestaurantId, RestaurantName, Restaurant\_Rating, Res\_location, Res\_Type, Monthly\_Income, Established\_Year of the different restaurants in the Tampa Location.

We have taken a simple table which consists of the 12 restaurants data. We have taken the rating information about the restaurant from the Zomato food delivery app. Below is the Restaurant table we have chosen.

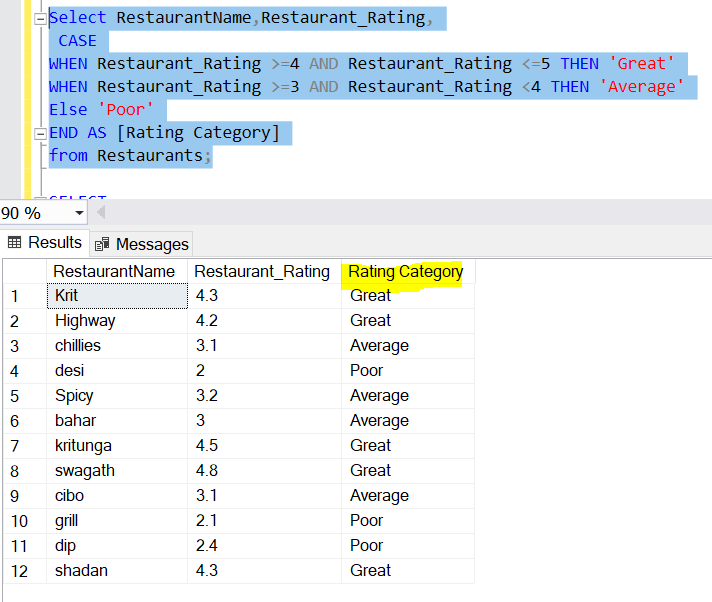
The table consists of the  Restaurant Id, Restaurant Name, rating column which is given by the users (on a scale of 1 to 5) from their experience, monthly income in dollars ranging between 2000 $ to 7000 $, restaurant’s established year, 4 different types of the style namely Mexican, Italian, Chinese and Indian and Location as shown in below screenshot.



**Multiple Conditions in CASE Statement:**

Now, we have be used the Case expression to evaluate multiple condition i.e. created a **[Rating Category**] column depending on the **Restaurant\_Rating**.

We have set the value of the Rating Category to “**Great**” if the Restaurant Rating is between 4 to 5, to “**Average**”, if the Restaurant rating is between 3 to 4 and if the restaurant rating is below 3 then Rating Category column is set to be “**Poor**” as shown in below snapshot.

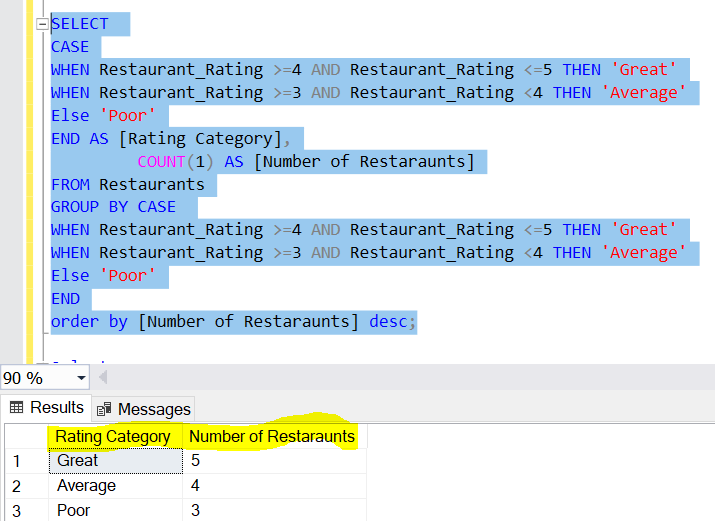


In the above query, if we see for the Kritunga restaurant,the Restaurant Rating is 4.5 which is above 4, so it falls in to the ‘Great’ Rating Category for the Restaurants in Tampa Area. Similiarly, we can explain about the other resaturants. To conclude, we have produced a new column ‘Rating Category’ based on the ‘Restaurant rating’ range of values.

**User’s Summary Table :**

Now, we want to know number of restaurants in the Tampa location having the Rating Category as Great, Average, Poor.

Here, we have used SQL Server CASE statement to set the value of the Rating Category column to ‘Great’, ‘Average’, ‘Poor’. Inside the GROUP BY clause, we specified the corresponding Count to be increased by 1, whenever it encounters the respective ‘Restaurant Rating’ column value. So, we have used the **Count()**  aggregate function to return the number of restaurants having the Rating category as Great, Average and Poor as shown in below screenshot



To conclude, the number of restaurants in Tampa location having the ‘Rating Category’ as Great is ‘5’, while the Average Rating Category is ‘4’ and Poor Category is ‘3’.

**Create 2 Sub queries: 1 correlated, 1 uncorrelated**

USER STORIES:

## ****Correlated Sub-query Example****

To retrieve the details of the employees whose department category is ‘Manager’

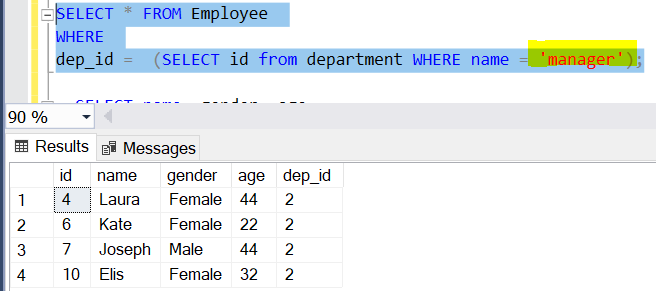
## ****Correlated Sub-query Example****

We want to know the Employee whose age is greater than the average age of all employees.

For this demonstration,We created a department and employee table as shown below

**Uncorrelated Sub-query Example**

Let us execute a uncorrelated sub-query which retrieves records of all the employees who belong to “Manager” Category.



## ****Correlated Sub-query Example****

We know that in case of correlated sub-queries, the inner query depends upon the outer query and cannot be executed as a standalone query.Lets execute a correlated sub-query that retrieves results of all the employees with age greater than average age within their department as discussed above.

