DATABASE MANGEMENT SYSTEMS

2016-2017 FALL SEMESTER

LABORATORY MANUAL

Experiment 5

Stored Procedure

The real power of stored procedures is the ability to pass parameters and have the stored procedure handle the differing requests that are made. Just like you have the ability to use parameters with your SQL code you can also setup your stored procedures to accept one or more parameter values.

- **One Parameter**

In this example we will query the Person.Address table from the AdventureWorks database, but instead of getting back all records we will limit it to just a particular city. This example assumes there will be an exact match on the City value that is passed.

```
CREATE PROCEDURE uspGetAddress @City nvarchar(30)
AS
SELECT *
FROM AdventureWorks.Person.Address
WHERE City = @City
GO
```

To call this stored procedure we would execute it as follows:

```
EXEC uspGetAddress @City = 'New York'
```

We can also do the same thing, but allow the users to give us a starting point to search the data. Here we can change the "=" to a LIKE and use the "%" wildcard.

```
CREATE PROCEDURE uspGetAddress @City nvarchar(30)
AS
SELECT *
FROM AdventureWorks.Person.Address
WHERE City LIKE @City + '%'
GO
```

In both of the proceeding examples it assumes that a parameter value will always be passed. If you try to execute the procedure without passing a parameter value you will get an error message such as the following:

```
Msg 201, Level 16, State 4, Procedure uspGetAddress, Line 0

Procedure or function 'uspGetAddress' expects parameter '@City', which was not
supplied.
```

- **Default Parameter Values**

In most cases it is always a good practice to pass in all parameter values, but sometimes it is not possible. So in this example we use the NULL option to allow you to not pass in a parameter value. If we create and run this stored procedure as is it will not return any data, because it is looking for any City values that equal NULL.

```
CREATE PROCEDURE uspGetAddress @City nvarchar(30) = NULL
AS
SELECT *
FROM AdventureWorks.Person.Address
WHERE City = @City
GO
```

We could change this stored procedure and use the ISNULL function to get around this. So if a value is passed it will use the value to narrow the result set and if a value is not passed it will return all records. (Note: if the City column has NULL values this will not include these values. You will have to add additional logic for City IS NULL)

```
CREATE PROCEDURE uspGetAddress @City nvarchar(30) = NULL
AS
SELECT *
FROM AdventureWorks.Person.Address
WHERE City = ISNULL(@City,City)
GO
```

- **Multiple Parameters**

Setting up multiple parameters is very easy to do. You just need to list each parameter and the data type separated by a comma as shown below.

```
CREATE PROCEDURE uspGetAddress @City nvarchar(30) = NULL, @AddressLine1 nvarchar(60) =
NULL
AS
SELECT *
FROM AdventureWorks.Person.Address
WHERE City = ISNULL(@City,City)
AND AddressLine1 LIKE '%' + ISNULL(@AddressLine1 ,AddressLine1) + '%'
GO
```

To execute this you could do any of the following:

```
EXEC uspGetAddress @City = 'Calgary'
--or
EXEC uspGetAddress @City = 'Calgary', @AddressLine1 = 'A'
--or
EXEC uspGetAddress @AddressLine1 = 'Acardia'
-- etc...
```

- **Dropping Stored Procedure**

To drop a single stored procedure you use the DROP PROCEDURE or DROP PROC command as follows.

```
DROP PROCEDURE uspGetAddress
GO
-- or
DROP PROC uspGetAddress
GO
-- or
DROP PROC dbo.uspGetAddress -- also specify the schema
```

- **Modifying an Existing Stored Procedure**

To change the stored procedure and save the updated code you would use the ALTER PROCEDURE command as follows.

```
ALTER PROCEDURE uspGetAddress @City nvarchar(30)
AS
SELECT *
FROM AdventureWorks.Person.Address
```

```sql
WHERE City LIKE @City + '%'
GO
```

View

In SQL, a view is a virtual table based on the result-set of an SQL statement. A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database. You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.

```sql
CREATE VIEW [Current Product List] AS
SELECT ProductID,ProductName
FROM Products
WHERE Discontinued=No
```

We can query the view above as follows:

```sql
SELECT * FROM [Current Product List]
```

Another view in the Northwind sample database selects every product in the "Products" table with a unit price higher than the average unit price:

```sql
CREATE VIEW [Products Above Average Price] AS
SELECT ProductName,UnitPrice
FROM Products
WHERE UnitPrice>(SELECT AVG(UnitPrice) FROM Products)
```

We can query the view above as follows:

```sql
SELECT * FROM [Products Above Average Price]
```

Another view in the Northwind database calculates the total sale for each category in 1997. Note that this view selects its data from another view called "Product Sales for 1997":

```sql
CREATE VIEW [Category Sales For 1997] AS
SELECT DISTINCT CategoryName,Sum(ProductSales) AS CategorySales
FROM [Product Sales for 1997]
GROUP BY CategoryName
```

We can query the view above as follows:

```
SELECT * FROM [Category Sales For 1997]
```

We can also add a condition to the query. Now we want to see the total sale only for the category "Beverages":

```
SELECT * FROM [Category Sales For 1997]
WHERE CategoryName='Beverages'
```

- **SQL Updating View**

Now we want to add the "Category" column to the "Current Product List" view. We will update the view with the following SQL:

```
CREATE OR REPLACE VIEW [Current Product List] AS
SELECT ProductID,ProductName,Category
FROM Products
WHERE Discontinued=No
```

- **SQL Dropping View**

You can delete a view with the DROP VIEW command.

```
DROP VIEW [Current Product List]
```

Function

There are three types of User-Defined functions in SQL Server and they are Scalar, Inline Table-Valued and Multi-statement Table-valued.

- **Scalar User-Defined Function**

A Scalar user-defined function returns one of the scalar data types. Text, ntext, image and timestamp data types are not supported. These are the type of user-defined functions that most developers are used to in other programming languages. You pass in 0 to many parameters and you get a return value. Below is an example that is based in the data found in the NorthWind Customers Table.

```
CREATE FUNCTION whichContinent
(@Country nvarchar(15))
RETURNS varchar(30)
AS
BEGIN
declare @Return varchar(30)
select @return = case @Country
when 'Argentina' then 'South America'
when 'Belgium' then 'Europe'
when 'Brazil' then 'South America'
when 'Canada' then 'North America'
when 'Denmark' then 'Europe'
when 'Finland' then 'Europe'
when 'France' then 'Europe'
else 'Unknown'
end

return @return
end
```

We can query the function above as follows:

```
select dbo.WhichContinent(Customers.Country)
```

- **Inline Table-Value User-Defined Function**

An Inline Table-Value user-defined function returns a table data type and is an exceptional alternative to a view as the user-defined function can pass parameters into a T-SQL select command and in essence provide us with a parameterized, non-updateable view of the underlying tables.

```
CREATE FUNCTION CustomersByContinent
(@Continent varchar(30))
RETURNS TABLE
AS
RETURN
       SELECT dbo.WhichContinent(Customers.Country) as continent, customers.*
       FROM customers
       WHERE dbo.WhichContinent(Customers.Country) = @Continent
GO
```

We can query the function above as follows:

```sql
SELECT * from CustomersbyContinent('North America')
```

- **Multi-Statement Table-Value User-Defined Function**

A Multi-Statement Table-Value user-defined function returns a table and is also an exceptional alternative to a view as the function can support multiple T-SQL statements to build the final result where the view is limited to a single SELECT statement. Also, the ability to pass parameters into a T-SQL select command or a group of them gives us the capability to in essence create a parameterized, non-updateable view of the data in the underlying tables. Within the create function command you must define the table structure that is being returned. After creating this type of user-defined function, I can use it in the FROM clause of a T-SQL command unlike the behavior found when using a stored procedure which can also return record sets.

```sql
CREATE FUNCTION dbo.customersbycountry ( @Country varchar(15) )
RETURNS
       @CustomersbyCountryTab table (
              [CustomerID] [nchar] (5),
              [CompanyName] [nvarchar] (40),
              [ContactName] [nvarchar] (30),
              [ContactTitle] [nvarchar] (30),
              [Address] [nvarchar] (60),
              [City] [nvarchar] (15),
              [PostalCode] [nvarchar] (10),
              [Country] [nvarchar] (15),
              [Phone] [nvarchar] (24),
              [Fax] [nvarchar] (24)
       )
AS
BEGIN
       INSERT INTO @CustomersByCountryTab
       SELECT [CustomerID],
              [CompanyName],
              [ContactName],
              [ContactTitle],
              [Address],
              [City],
              [PostalCode],
                [Country],
                [Phone],
                [Fax]
        FROM [Northwind].[dbo].[Customers]
        WHERE country = @Country

        DECLARE @cnt INT
        SELECT @cnt = COUNT(*) FROM @customersbyCountryTab

        IF @cnt = 0
                INSERT INTO @CustomersByCountryTab (
                        [CustomerID],
                        [CompanyName],
                        [ContactName],
                        [ContactTitle],
                        [Address],
                        [City],
                        [PostalCode],
                        [Country],
                        [Phone],
                        [Fax] )
                VALUES ('','No Companies Found','','','','','','','','')
        RETURN
END
GO

SELECT * FROM dbo.customersbycountry('USA')
```

- **Drop Function**

```
DROP FUNCTION dbo.customersbycountry;
```

<u>Exercises :</u>

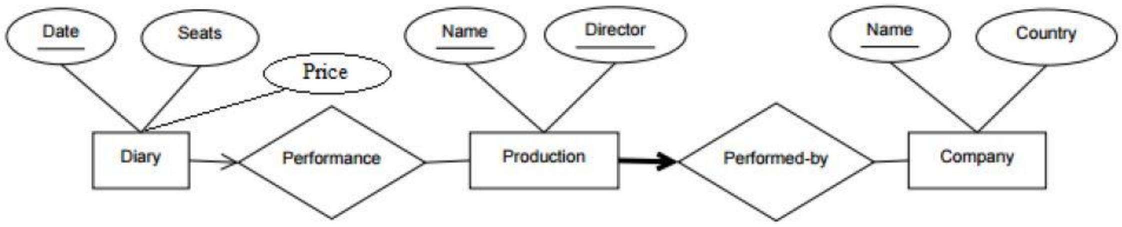

Fig. 1. The ER diagram for a theatre manager database

Theatre manager database ER diagram is given in Fig. 1. This database keeps information of forthcoming shows by acting companies. Answer following questions using SQL according to this database.

1. Create database named as “db_TheatreManager” .
2. Write a stored procedure to find the max price of production performed by company named as ‘A’. Use company name as an input parameter for the stored procedure.
3. Write a function to find the number of productions performed by company named as ‘B’. Use company name as an input parameter for the function.
4. Write a view that select dates of production performed by company ‘A’.
5. Write a function to find the name of production performaed by company ‘A’ and country in ‘Eskişehir’.
6. Write a stored procedure to find the prices of all production whose director is ‘X’.
7. Write a function to find all production that are performed in ‘Ankara’.
8. Write a stored procedure to select name and director of all production that is performanced on ’01.12.2016’ and price is smaller 100 YTL.
9. Update the stored procedure in question 6 to accept date also an input parameter.
10. Update the view in question 4 to accept director also an input parameter.