Store Procedure

A stored procedure is a group of sql statements that has been created and stored in the database. Stored procedure will accept input parameters so that a single procedure can be used over the network by several clients using different input data. Stored procedure will reduce network traffic and increase the performance. If we modify stored procedure all the clients will get the updated stored procedure

**Sample of creating Stored Procedure**

|  |
| --- |
| USE AdventureWorks2008R2;  GO  CREATE PROCEDURE dbo.sp\_who  AS      SELECT FirstName, LastName FROM Person.Person;  GO  EXEC sp\_who;  EXEC dbo.sp\_who;  GO  DROP PROCEDURE dbo.sp\_who;  GO |

**Advantages of using stored procedures**

a)    a) Stored procedure allows modular programming.

You can create the procedure once, store it in the database, and call it any number of times in your program.

b)    b) Stored Procedure allows faster execution.

If the operation requires a large amount of SQL code is performed repetitively, stored procedures can be faster. They are parsed and optimized when they are first executed, and a compiled version of the stored procedure remains in memory cache for later use. This means the stored procedure does not need to be reparsed and reoptimized with each use resulting in much faster execution times.

c)     c) Stored Procedure can reduce network traffic.

An operation requiring hundreds of lines of Transact-SQL code can be performed through a single statement that executes the code in a procedure, rather than by sending hundreds of lines of code over the network.

d)    d) Stored procedures provide better security to your data

Users can be granted permission to execute a stored procedure even if they do not have permission to execute the procedure's statements directly.

In SQL we are having different types of stored procedures are there

a)    System Stored Procedures

b)    User Defined Stored procedures

c)    Extended Stored Procedures

**System Stored Procedures:**

System stored procedures are stored in the master database and these are starts with a **sp\_**prefix. These procedures can be used to perform variety of tasks to support sql server functions for external application calls in the system tables

Ex: sp\_helptext [StoredProcedure\_Name]

**User Defined Stored Procedures:**

User Defined stored procedures are usually stored in a user database and are typically designed to complete the tasks in the user database. While coding these procedures don’t use **sp\_**prefixbecauseif we use the **sp\_**prefix first it will check master database then it comes to user defined database

**Extended Stored Procedures:**

Extended stored procedures are the procedures that call functions from DLL files. Now a day’s extended stored procedures are depreciated for that reason it would be better to avoid using of Extended Stored procedures.

@@IDENTITY

It returns the last identity value generated for any table in the current session, across all scopes.

Let me explain this... suppose we create an insert trigger on table which inserts a row in another table with generate an identity column, then @@IDENTITY returns that identity record which is created by trigger.

SCOPE\_IDENTITY

It returns the last identity value generated for any table in the current session and the current scope.

Let me explain this... suppose we create an insert trigger on table which inserts a row in another table with generate an identity column, then SCOPE\_IDENTITY result is not affected but if a trigger or a user defined function is affected on the same table that produced the value returns that identity record thenSCOPE\_IDENTITY returns that identity record which is created by trigger or a user defined function.

IDENT\_CURRENT

It returns the last identity value generated for a specific table in any session and any scope.

In other words, we can say it is not affected by scope and session, it only depends on a particular table and returns that table related identity value which is generated in any session or scope.

[Returning Data Using an Output Parameter](javascript:void(0))

If you specify the OUTPUT keyword for a parameter in the procedure definition, the procedure can return the current value of the parameter to the calling program when the procedure exits. To save the value of the parameter in a variable that can be used in the calling program, the calling program must use the OUTPUT keyword when executing the procedure. For more information about what data types can be used as output parameters, see [CREATE PROCEDURE (Transact-SQL)](http://technet.microsoft.com/en-us/library/ms187926.aspx).

Examples of Output Parameter

The following example shows a procedure with an input and an output parameter. The @SalesPerson parameter would receive an input value specified by the calling program. The SELECT statement uses the value passed into the input parameter to obtain the correct SalesYTD value. The SELECT statement also assigns the value to the @SalesYTD output parameter, which returns the value to the calling program when the procedure exits.

Transact-SQL

USE AdventureWorks2012;

GO

IF OBJECT\_ID('Sales.uspGetEmployeeSalesYTD', 'P') IS NOT NULL

DROP PROCEDURE Sales.uspGetEmployeeSalesYTD;

GO

CREATE PROCEDURE Sales.uspGetEmployeeSalesYTD

@SalesPerson nvarchar(50),

@SalesYTD money OUTPUT.

AS

SET NOCOUNT ON;

SELECT @SalesYTD = SalesYTD

FROM Sales.SalesPerson AS sp

JOIN HumanResources.vEmployee AS e ON e.BusinessEntityID = sp.BusinessEntityID

WHERE LastName = @SalesPerson;

RETURN

GO

Transact-SQL

-- Declare the variable to receive the output value of the procedure.

DECLARE @SalesYTDBySalesPerson money;

-- Execute the procedure specifying a last name for the input parameter

-- and saving the output value in the variable @SalesYTDBySalesPerson

EXECUTE Sales.uspGetEmployeeSalesYTD

N'Blythe', @SalesYTD = @SalesYTDBySalesPerson OUTPUT;

-- Display the value returned by the procedure.

PRINT 'Year-to-date sales for this employee is ' +

convert(varchar(10),@SalesYTDBySalesPerson);

GO

Input values can also be specified for OUTPUT parameters when the procedure is executed. This allows the procedure to receive a value from the calling program, change or perform operations with the value, and then return the new value to the calling program. In the previous example, the @SalesYTDBySalesPerson variable can be assigned a value before the program calls the Sales.uspGetEmployeeSalesYTD procedure. The execute statement would pass the @SalesYTDBySalesPerson variable value into the @SalesYTD OUTPUT parameter. Then in the procedure body, the value could be used for calculations that generate a new value. The new value would be passed back out of the procedure through the OUTPUT parameter, updating the value in the @SalesYTDBySalesPerson variable when the procedure exits. This is often referred to as "pass-by-reference capability."

If you specify OUTPUT for a parameter when you call a procedure and that parameter is not defined by using OUTPUT in the procedure definition, you get an error message. However, you can execute a procedure with output parameters and not specify OUTPUT when executing the procedure. No error is returned, but you cannot use the output value in the calling program.

#### Stored Procedures - Output Parameters & Return Values

See Also: [Main\_Page](http://www.toadworld.com/platforms/sql-server/w/wiki/9386.sql-server-topics.aspx) - [Transact SQL Coding Techniques](http://www.toadworld.com/platforms/sql-server/w/wiki/10086.transact-sql-coding-techniques.aspx) - [Reusable Coding Techniques](http://www.toadworld.com/platforms/sql-server/w/wiki/10222.reusable-coding-techniques.aspx) - [Stored Procedures](http://www.toadworld.com/platforms/sql-server/w/wiki/10256.stored-procedures.aspx)

Besides using input parameters, stored procedures can also return OUTPUT parameters and return values. Output parameters behave similarly to input parameters, but have to be declared with the OUTPUT keyword. In addition, you should specify the OUTPUT keyword when executing a stored procedure containing the output parameter to get the value. The following procedure contains an input parameter of the title type and an output parameter of total quantity of titles sold for the specified type:

CREATE PROC sales\_for\_type @type VARCHAR(55), @total\_sales INT OUTPUT

AS

SELECT SUM(qty) FROM sales a, titles b

WHERE

a.title\_id = b.title\_id

and

b.type = @type

This procedure can be executed as follows:

DECLARE @total\_sales\_business int

EXEC sales\_for\_type business, @total\_sales=@total\_sales\_business OUTPUT

Results:

90

Notice that in order to use the output parameter, we have to declare a variable with the same data type as the output parameter of the called stored procedure. We can easily extend the same procedure to return more than one output parameters:

ALTER PROC sales\_for\_type @type VARCHAR(55), @total\_sales INT OUTPUT, @avg\_sales INT OUTPUT

AS

SELECT SUM(qty), AVG(qty) FROM sales a, titles b

WHERE

a.title\_id = b.title\_id

and

b.type = @type

Now, we can execute the new procedure as follows:

DECLARE @total\_sales\_business INT, @avg\_sales\_business INT

EXEC sales\_for\_type business, @total\_sales=@total\_sales\_business OUTPUT,

@avg\_sales = @avg\_sales\_business OUTPUT

Results:

-----------

90 18

You can execute the stored procedure with an output parameter without the OUTPUT keyword, but you won't be able to use the returned value in the calling program.

Return values can be used within stored procedures to provide the stored procedure execution status to the calling program. The return values -99 through 0 are reserved for SQL Server internal use. You can create your own parameters that can be passed back to the calling program. By default, the successful execution of a stored procedure (or any group of SQL statements) will return 0. The syntax of the return command is:

RETURN integer\_value

You can check the result of executing a stored procedures with return values as follows:

EXEC @return\_variable = stored\_procedure\_name

where **@return\_variable** is a numeric variable used to check the return value.

You can optionally enclose the integer value in parenthesis. If you don't supply the integer value, SQL Server will provide a value for you, depending on the state of program execution. RETURN also unconditionally exits the program, so once a RETURN is encountered in your T-SQL code SQL Server will not check any other conditions.

The following example demonstrates usage of user-defined return codes; notice that even though multiple conditions are examined within the procedure, a single RETURN will cause the program to stop and return the appropriate value:

ALTER PROC sales\_for\_type @type VARCHAR(55), @total\_sales INT OUTPUT, @avg\_sales INT OUTPUT

AS

IF @type IS NULL

BEGIN PRINT 'type is required' RETURN (1)

END

SELECT @total\_sales=SUM(qty), @avg\_sales = AVG(qty) FROM sales a, titles b

WHERE

a.title\_id = b.title\_id

and

b.type = @type

IF @total\_sales IS NULL

AND @avg\_sales IS NULL

BEGIN

RETURN (3) -- both avg and sum are null

END

IF @avg\_sales IS NULL

BEGIN

RETURN (1) -- avg is null

END

IF @total\_sales IS NULL

BEGIN

RETURN (2) -- total is null

END

Now, we can execute the procedure with intentionally wrong values to check how the RETURN statement works:

DECLARE @total\_sales\_business INT, @avg\_sales\_business INT, @return\_status INT

EXEC @return\_status = sales\_for\_type tomato, @total\_sales=@total\_sales\_business OUTPUT,

@avg\_sales = @avg\_sales\_business OUTPUT

SELECT @return\_status

Result:

3

DECLARE @total\_sales\_business INT, @avg\_sales\_business INT, @return\_status INT

EXEC @return\_status = sales\_for\_type NULL, @total\_sales=@total\_sales\_business OUTPUT,

@avg\_sales = @avg\_sales\_business OUTPUT

SELECT @return\_status

Result:

type is required

1

Retrun value

ParameterDirection.ReturnValue should be used for the "return value" of the procedure, not output parameters. It gets the value returned by the SQL RETURN statement (with the parameter named @RETURN\_VALUE).

Instead of RETURN @b you should SET @b = something

By the way, return value parameter is always int, not string.

ALTER procedure [dbo].[change\_password]

(

@user\_id nvarchar(10),

@password nvarchar(100),

@newpassword nvarchar(100)

)

as

begin

if (select count(UserId) from UserMst where [Password] = @password and UserId = @user\_id and IsActive ='Y') > 0

begin

update UserMst set Password = @newpassword where UserId = @user\_id

select 1

end

else

begin

select 0

end

end

result = Convert.ToInt32(objoperation.ExecuteScalar(spName, alParameter,transaction));

SET ANSI\_NULLS ON

GO

SET QUOTED\_IDENTIFIER ON

GO

alter PROCEDURE get\_output\_return\_val

@sem\_code varchar(10),

@year\_code varchar(10),

@user\_id varchar(10),

@output varchar(100) out

AS

BEGIN

-- SET NOCOUNT ON added to prevent extra result sets from

-- interfering with SELECT statements.

SET NOCOUNT ON;

-- Insert statements for procedure here

--set @output = (SELECT @sem\_code +' - ' + @year\_code + ' - ' + @user\_id)

set @output = (SELECT @sem\_code +' - ' + @year\_code + ' - ' + @user\_id)

select \* from user\_mst

return @@rowcount

END

GO

cmd.Parameters.Clear();

cmd.Parameters.AddWithValue("@sem\_code", obj\_data["sem\_code"]);

cmd.Parameters.Add("@year\_code", SqlDbType.VarChar).Value = obj\_data["year\_code"];

// cmd.Parameters.Add("@user\_id", SqlDbType.VarChar).Value = obj\_data["user\_id"];

cmd.Parameters.Add(new SqlParameter("@user\_id", obj\_data["user\_id"]));

SqlParameter returnValue = new SqlParameter();

returnValue.Direction = ParameterDirection.ReturnValue;

cmd.Parameters.Add(returnValue);

cmd.Parameters.Add("@output",SqlDbType.VarChar,100).Direction = ParameterDirection.output ;

cmd.Parameters.Add("@ReturnVal", SqlDbType.Int).Direction = ParameterDirection.ReturnValue;

// cmd.Parameters.Add("@output\_val", SqlDbType.VarChar, 30);

// cmd.Parameters["@output\_val"].Direction = ParameterDirection.Output;

//SqlParameter returnValue = new SqlParameter("return\_val",SqlDbType.VarChar,30);

//returnValue.Direction = ParameterDirection.ReturnValue;

//cmd.Parameters.Add(returnValue);

// cmd.Transaction = con.BeginTransaction();

adp = new SqlDataAdapter(cmd);

cmd.Connection = con;

cmd.CommandText = "get\_output\_return\_val";

cmd.CommandType = CommandType.StoredProcedure;

DataSet ds = new DataSet();

try

{

adp.Fill(ds);

//Get retunn value from SP

//1st

int aaaa = returnValue.Value;

//2nd

int return\_value = (int)cmd.Parameters["@ReturnVal"].Value;

string output\_val = cmd.Parameters["@output"].ToString();

if (ds.Tables[0].Rows.Count <= 0)

return null;

else

return ds.Tables[0];

}

catch (Exception ex)

{

return null;

}

return null;

=== Also use with another

cmd.Parameters.Add("@ReturnVal", SqlDbType.Int).Direction = ParameterDirection.ReturnValue;

(int)cmd.Parameters["@ReturnVal"].Value;

# Grant Permissions on a Stored Procedure

#### ermissions

The grantor (or the principal specified with the AS option) must have either the permission itself with GRANT OPTION, or a higher permission that implies the permission being granted. Requires ALTER permission on the schema to which the procedure belongs, or CONTROL permission on the procedure. For more information, see [GRANT Object Permissions (Transact-SQL)](https://msdn.microsoft.com/en-us/library/ms188371.aspx).

[[Top]](https://msdn.microsoft.com/en-us/library/ms345484.aspx#Intro)

## [Using SQL Server Management Studio](javascript:void(0))

### To grant permissions on a stored procedure

1. In Object Explorer, connect to an instance of Database Engine and then expand that instance.
2. Expand **Databases**, expand the database in which the procedure belongs, and then expand **Programmability**.
3. Expand **Stored Procedures**, right-click the procedure to grant permissions on, and then click **Properties**.
4. From **Stored Procedure Properties**, select the **Permissions** page.
5. To grant permissions to a user, database role, or application role, click **Search**.
6. In **Select Users or Roles**, click **Object Types** to add or clear the users and roles you want.
7. Click **Browse** to display the list of users or roles. Select the users or roles to whom permissions should be granted.
8. In the **Explicit Permissions** grid, select the permissions to grant to the specified user or role. For a description of the permissions, see [Permissions (Database Engine)](https://msdn.microsoft.com/en-us/library/ms191291.aspx).

Selecting **Grant** indicates the grantee will be given the specified permission. Selecting **Grant With** indicates that the grantee will also be able to grant the specified permission to other principals.

[[Top]](https://msdn.microsoft.com/en-us/library/ms345484.aspx#Intro)

## [Using Transact-SQL](javascript:void(0))

### To grant permissions on a stored procedure

1. Connect to the Database Engine.
2. From the Standard bar, click **New Query**.
3. Copy and paste the following example into the query window and click **Execute**. This example grants EXECUTE permission on the stored procedureHumanResources.uspUpdateEmployeeHireInfo to an application role named Recruiting11.

Transact-SQL

USE AdventureWorks2012;

GRANT EXECUTE ON OBJECT::HumanResources.uspUpdateEmployeeHireInfo

TO Recruiting11;

GO

Type of Store Proc

**Types Of Stored  Procedures**

* User Defined Stored procedure:The user defined stored procedures are created by users and stored in the current database
* System Stored Procedure: The system stored procedure have names prefixed with sp\_. Its manage SQL Server through administrative tasks. Which databases store system stored procedures are master and msdb database
* Temporary Stored procedures: The temporary stored procedures have names prefixed with the # symbol. Temporary stored procedures stored in the tempdb databases. These procedures are automatically dropped when the connection  terminates between client and server
* Remote Stored Procedures: The remote stored procedures are procedures that are created and stored in databases on remote servers. These remote procedures can be accessed from various servers, provided the users have the appropriate permission
* Extended Stored Procedures: These are Dynamic-link libraries (DLL's) that are executed outside the SQL Server environment. They are identified by the prefix xp\_

Extended procedures provide an interface to external programs for various maintenance activities. These extended procedures starts with the xp\_ prefix and stored in Master database. Basically these are used to call programs that reside on the server automatically from a stored procedure or a trigger run by the server.

**Example**Below statements are used to log an event in the NT event log of the server without raising any error on the client application.

1. **declare @logmsg varchar(100)**
2. **set @logmsg = suser\_sname() + ': Tried to access the dotnet system.'**
3. **exec xp\_logevent 50005, @logmsg**
4. **print @logmsg**

**Example**The below procedure will display details about the BUILTIN\Administrators Windows group.

1. **EXEC xp\_logininfo 'BUILTIN\Administrators'**

## User Defined Stored Procedure

These procedures are created by user for own actions. These can be created in all system databases except the Resource database or in a user-defined database.

## CLR Stored Procedure

CLR stored procedure are special type of procedure that are based on the CLR (Common Language Runtime) in .net framework. CLR integration of procedure was introduced with SQL Server 2008 and allow for procedure to be coded in one of .NET languages like C#, Visual Basic and F#. I will discuss CLR stored procedure later.