









CREATE PROCEDURE (Transact-SQL)

Applies to:  [SQL Server](#)  [Azure SQL Database](#)  [Azure SQL Managed Instance](#)  [Azure Synapse Analytics](#)  [Analytics Platform System \(PDW\)](#)  [SQL analytics endpoint in Microsoft Fabric](#)  [Warehouse in Microsoft Fabric](#)  [SQL database in Microsoft Fabric Preview](#)

Creates a Transact-SQL or common language runtime (CLR) stored procedure in SQL Server, Azure SQL Database, SQL database in Microsoft Fabric Preview, and Analytics Platform System (PDW). Stored procedures are similar to procedures in other programming languages in that they can:

- Accept input parameters and return multiple values in the form of output parameters to the calling procedure or batch.
- Contain programming statements that perform operations in the database, including calling other procedures.
- Return a status value to a calling procedure or batch to indicate success or failure (and the reason for failure).

Use this statement to create a permanent procedure in the current database or a temporary procedure in the `tempdb` database.

Note

The integration of .NET Framework CLR into SQL Server is discussed in this topic. CLR integration does not apply to Azure SQL Database or SQL database in Microsoft Fabric Preview.

Jump to [Simple Examples](#) to skip the details of the syntax and get to a quick example of a basic stored procedure.

 [Transact-SQL syntax conventions](#)

Syntax

Transact-SQL syntax for stored procedures in SQL Server, Azure SQL Database, SQL database in Microsoft Fabric Preview:

```
syntaxsql
```

```
CREATE [ OR ALTER ] { PROC | PROCEDURE }  
    [ schema_name. ] procedure_name [ ; number ]
```

```

    [ { @parameter_name [ type_schema_name. ] data_type }
      [ VARYING ] [ NULL ] [ = default ] [ OUT | OUTPUT | [READONLY]
    ] [ ,...n ]
[ WITH <procedure_option> [ ,...n ] ]
[ FOR REPLICATION ]
AS { [ BEGIN ] sql_statement [;] [ ...n ] [ END ] }
[;]

<procedure_option> ::=
    [ ENCRYPTION ]
    [ RECOMPILE ]
    [ EXECUTE AS clause ]

```

Transact-SQL syntax for CLR stored procedures:

syntaxsql

```

CREATE [ OR ALTER ] { PROC | PROCEDURE }
    [ schema_name. ] procedure_name [ ; number ]
    [ { @parameter_name [ type_schema_name. ] data_type }
      [ = default ] [ OUT | OUTPUT ] [ READONLY ]
    ] [ ,...n ]
[ WITH EXECUTE AS clause ]
AS { EXTERNAL NAME assembly_name.class_name.method_name }
[;]

```

Transact-SQL syntax for natively compiled stored procedures:

syntaxsql

```

CREATE [ OR ALTER ] { PROC | PROCEDURE } [ schema_name. ] procedure_name
    [ { @parameter data_type } [ NULL | NOT NULL ] [ = default ]
      [ OUT | OUTPUT ] [ READONLY ]
    ] [ ,... n ]
    WITH NATIVE_COMPILATION, SCHEMABINDING [ , EXECUTE AS clause ]
AS
{
    BEGIN ATOMIC WITH ( <set_option> [ ,... n ] )
sql_statement [;] [ ... n ]
[ END ]
}
[;]

<set_option> ::=
    LANGUAGE = [ N ] 'language'
    | TRANSACTION ISOLATION LEVEL = { SNAPSHOT | REPEATABLE READ | SERIALIZABLE }
    | [ DATEFIRST = number ]

```

```
| [ DATEFORMAT = format ]  
| [ DELAYED_DURABILITY = { OFF | ON } ]
```

Transact-SQL syntax for stored procedures in Azure Synapse Analytics and Parallel Data Warehouse:

syntaxsql

```
CREATE { PROC | PROCEDURE } [ schema_name.] procedure_name  
    [ { @parameter data_type } [ OUT | OUTPUT ] ] [ ,...n ]  
AS  
{  
    [ BEGIN ] sql_statement [;][ ,...n ] [ END ]  
}  
[;]
```

Transact-SQL syntax for stored procedures in Microsoft Fabric:

syntaxsql

```
CREATE [ OR ALTER ] { PROC | PROCEDURE } [ schema_name.] procedure_name  
    [ { @parameter data_type } [ OUT | OUTPUT ] ] [ ,...n ]  
AS  
{  
    [ BEGIN ] sql_statement [;][ ,...n ] [ END ]  
}  
[;]
```

Arguments

OR ALTER

Applies to: Azure SQL Database, SQL database in Microsoft Fabric Preview, SQL Server (starting with SQL Server 2016 (13.x) SP1).

Alters the procedure if it already exists.

schema_name

The name of the schema to which the procedure belongs. Procedures are schema-bound. If a schema name isn't specified when the procedure is created, the default schema of the user who is creating the procedure is automatically assigned.

procedure_name

The name of the procedure. Procedure names must comply with the rules for [identifiers](#) and must be unique within the schema.

Caution

Avoid the use of the `sp_` prefix when naming procedures. This prefix is used by SQL Server to designate system procedures. Using the prefix can cause application code to break if there is a system procedure with the same name.

Local or global temporary procedures can be created by using one number sign (#) before *procedure_name* (*#procedure_name*) for local temporary procedures, and two number signs for global temporary procedures (*##procedure_name*). A local temporary procedure is visible only to the connection that created it and is dropped when that connection is closed. A global temporary procedure is available to all connections and is dropped at the end of the last session using the procedure. Temporary names can't be specified for CLR procedures.

The complete name for a procedure or a global temporary procedure, including `##`, can't exceed 128 characters. The complete name for a local temporary procedure, including `#`, can't exceed 116 characters.

; number

Applies to: SQL Server 2008 (10.0.x) and later versions, Azure SQL Database, SQL database in Microsoft Fabric Preview.

An optional integer that is used to group procedures of the same name. These grouped procedures can be dropped together by using one `DROP PROCEDURE` statement.

Note

This feature will be removed in a future version of SQL Server. Avoid using this feature in new development work, and plan to modify applications that currently use this feature.

Numbered procedures can't use the `xml` or CLR user-defined types and can't be used in a plan guide.

@*parameter_name*

A parameter declared in the procedure. Specify a parameter name by using the at sign (@) as the first character. The parameter name must comply with the rules for [identifiers](#). Parameters are local to the procedure; the same parameter names can be used in other procedures.

One or more parameters can be declared; the maximum is 2,100. The value of each declared parameter must be supplied by the user when the procedure is called unless a default value for the parameter is defined or the value is set to equal another parameter. If a procedure contains [table-valued parameters](#), and the parameter is missing in the call, an empty table is passed in. Parameters can take the place only of constant expressions; they can't be used instead of table names, column names, or the names of other database objects. For more information, see [EXECUTE \(Transact-SQL\)](#).

Parameters can't be declared if FOR REPLICATION is specified.

[*type_schema_name*.] *data_type*

The data type of the parameter and the schema to which the data type belongs.

Guidelines for Transact-SQL procedures:

- All Transact-SQL data types can be used as parameters.
- You can use the user-defined table type to create table-valued parameters. Table-valued parameters can only be INPUT parameters and must be accompanied by the READONLY keyword. For more information, see [Use Table-Valued Parameters \(Database Engine\)](#)
- **cursor** data types can only be OUTPUT parameters and must be accompanied by the VARYING keyword.

Guidelines for CLR procedures:

- All of the native SQL Server data types that have an equivalent in managed code can be used as parameters. For more information about the correspondence between CLR types and SQL Server system data types, see [Mapping CLR Parameter Data](#). For more information about SQL Server system data types and their syntax, see [Data Types \(Transact-SQL\)](#).
- Table-valued or **cursor** data types can't be used as parameters.
- If the data type of the parameter is a CLR user-defined type, you must have EXECUTE permission on the type.

VARYING

Specifies the result set supported as an output parameter. This parameter is dynamically constructed by the procedure and its contents may vary. Applies only to **cursor** parameters. This option isn't valid for CLR procedures.

default

A default value for a parameter. If a default value is defined for a parameter, the procedure can be executed without specifying a value for that parameter. The default value must be a constant or it can be NULL. The constant value can be in the form of a wildcard, making it possible to use the LIKE keyword when passing the parameter into the procedure.

Default values are recorded in the `sys.parameters.default` column only for CLR procedures. That column is NULL for Transact-SQL procedure parameters.

OUT | OUTPUT

Indicates that the parameter is an output parameter. Use OUTPUT parameters to return values to the caller of the procedure. **text**, **ntext**, and **image** parameters can't be used as OUTPUT parameters, unless the procedure is a CLR procedure. An output parameter can be a cursor placeholder, unless the procedure is a CLR procedure. A table-value data type can't be specified as an OUTPUT parameter of a procedure.

READONLY

Indicates that the parameter can't be updated or modified within the body of the procedure. If the parameter type is a table-value type, READONLY must be specified.

RECOMPILE

Indicates that the Database Engine doesn't cache a query plan for this procedure, forcing it to be compiled each time it is executed. For more information regarding the reasons for forcing a recompile, see [Recompile a Stored Procedure](#). This option can't be used when FOR REPLICATION is specified or for CLR procedures.

To instruct the Database Engine to discard query plans for individual queries inside a procedure, use the RECOMPILE query hint in the definition of the query. For more information, see [Query Hints \(Transact-SQL\)](#).

ENCRYPTION

Applies to: SQL Server 2008 (10.0.x) and later versions, Azure SQL Database, SQL database in Microsoft Fabric Preview.

Indicates that SQL Server converts the original text of the CREATE PROCEDURE statement to an obfuscated format. The output of the obfuscation isn't directly visible in any of the catalog views in SQL Server. Users who have no access to system tables or database files can't retrieve the obfuscated text. However, the text is available to privileged users who can either access system tables over the [DAC port](#) or directly access database files. Also, users who can attach a debugger to the server process can retrieve the decrypted procedure from memory at runtime. For more information about accessing system metadata, see [Metadata Visibility Configuration](#).

This option isn't valid for CLR procedures.

Procedures created with this option can't be published as part of SQL Server replication.

EXECUTE AS *clause*

Specifies the security context under which to execute the procedure.

For natively compiled stored procedures, there are no limitations on the EXECUTE AS clause. In SQL Server 2014 (12.x) and earlier versions, the SELF, OWNER, and 'user_name' clauses are supported with natively compiled stored procedures.

For more information, see [EXECUTE AS Clause \(Transact-SQL\)](#).

FOR REPLICATION

Applies to: SQL Server 2008 (10.0.x) and later versions, Azure SQL Database, SQL database in Microsoft Fabric Preview.

Specifies that the procedure is created for replication. Consequently, it can't be executed on the Subscriber. A procedure created with the FOR REPLICATION option is used as a procedure filter and is executed only during replication. Parameters can't be declared if FOR REPLICATION is specified. FOR REPLICATION can't be specified for CLR procedures. The RECOMPILE option is ignored for procedures created with FOR REPLICATION.

A FOR REPLICATION procedure has an object type **RF** in sys.objects and sys.procedures.

{ [BEGIN] *sql_statement* [;] [...*n*] [END] }

One or more Transact-SQL statements comprising the body of the procedure. You can use the optional BEGIN and END keywords to enclose the statements. For information, see the Best Practices, General Remarks, and Limitations and Restrictions sections that follow.

EXTERNAL NAME *assembly_name.class_name.method_name*

Applies to: SQL Server 2008 (10.0.x) and later versions, Azure SQL Database, SQL database in Microsoft Fabric Preview.

Specifies the method of a .NET Framework assembly for a CLR procedure to reference. *class_name* must be a valid SQL Server identifier and must exist as a class in the assembly. If the class has a namespace-qualified name that uses a period (.) to separate namespace parts, the class name must be delimited by using brackets ([]) or quotation marks (" "). The specified method must be a static method of the class.

By default, SQL Server can't execute CLR code. You can create, modify, and drop database objects that reference common language runtime modules; however, you can't execute these references in SQL Server until you enable the [clr enabled option](#). To enable the option, use [sp_configure](#).

Note

CLR procedures are not supported in a contained database.

ATOMIC WITH

Applies to: SQL Server 2014 (12.x) and later versions, Azure SQL Database, SQL database in Microsoft Fabric Preview.

Indicates atomic stored procedure execution. Changes are either committed or all of the changes rolled back by throwing an exception. The ATOMIC WITH block is required for natively compiled stored procedures.

If the procedure RETURNS (explicitly through the RETURN statement, or implicitly by completing execution), the work performed by the procedure is committed. If the procedure THROWS, the work performed by the procedure is rolled back.

XACT_ABORT is ON by default inside an atomic block and can't be changed. XACT_ABORT specifies whether SQL Server automatically rolls back the current transaction when a Transact-SQL statement raises a run-time error.

The following SET options are always ON in the ATOMIC block, and can't be changed.

- CONCAT_NULL_YIELDS_NULL
- QUOTED_IDENTIFIER, ARITHABORT
- NOCOUNT
- ANSI_NULLS
- ANSI_WARNINGS

SET options can't be changed inside ATOMIC blocks. The SET options in the user session aren't used in the scope of natively compiled stored procedures. These options are fixed at compile time.

BEGIN, ROLLBACK, and COMMIT operations can't be used inside an atomic block.

There is one ATOMIC block per natively compiled stored procedure, at the outer scope of the procedure. The blocks can't be nested. For more information about atomic blocks, see [Natively Compiled Stored Procedures](#).

NULL | NOT NULL

Determines whether null values are allowed in a parameter. NULL is the default.

NATIVE_COMPILATION

Applies to: SQL Server 2014 (12.x) and later versions, Azure SQL Database, SQL database in Microsoft Fabric Preview.

Indicates that the procedure is natively compiled. NATIVE_COMPILATION, SCHEMABINDING, and EXECUTE AS can be specified in any order. For more information, see [Natively Compiled Stored Procedures](#).

SCHEMABINDING

Applies to: SQL Server 2014 (12.x) and later versions, Azure SQL Database, SQL database in Microsoft Fabric Preview.

Ensures that tables that are referenced by a procedure can't be dropped or altered. SCHEMABINDING is required in natively compiled stored procedures. (For more information, see [Natively Compiled Stored Procedures](#).) The SCHEMABINDING restrictions are the same as they are for user-defined functions. For more information, see the SCHEMABINDING section in [CREATE FUNCTION \(Transact-SQL\)](#).

LANGUAGE = [N] 'language'

Applies to: SQL Server 2014 (12.x) and later versions, Azure SQL Database, SQL database in Microsoft Fabric Preview.

Equivalent to [SET LANGUAGE \(Transact-SQL\)](#) session option. LANGUAGE = [N] 'language' is required.

TRANSACTION ISOLATION LEVEL

Applies to: SQL Server 2014 (12.x) and later versions, Azure SQL Database, SQL database in Microsoft Fabric Preview.

Required for natively compiled stored procedures. Specifies the transaction isolation level for the stored procedure. The options are as follows:

For more information about these options, see [SET TRANSACTION ISOLATION LEVEL \(Transact-SQL\)](#).

REPEATABLE READ

Specifies that statements can't read data that has been modified but not yet committed by other transactions. If another transaction modifies data that has been read by the current transaction, the current transaction fails.

SERIALIZABLE

Specifies the following:

- Statements can't read data that has been modified but not yet committed by other transactions.
- If another transaction modifies data that has been read by the current transaction, the current transaction fails.

- If another transaction inserts new rows with key values that would fall in the range of keys read by any statements in the current transaction, the current transaction fails.

SNAPSHOT

Specifies that data read by any statement in a transaction is the transactionally consistent version of the data that existed at the start of the transaction.

DATEFIRST = *number*

Applies to: SQL Server 2014 (12.x) and later versions, Azure SQL Database, SQL database in Microsoft Fabric Preview.

Specifies the first day of the week to a number from 1 through 7. DATEFIRST is optional. If it isn't specified, the setting is inferred from the specified language.

For more information, see [SET DATEFIRST \(Transact-SQL\)](#).

DATEFORMAT = *format*

Applies to: SQL Server 2014 (12.x) and later versions, Azure SQL Database, SQL database in Microsoft Fabric Preview.

Specifies the order of the month, day, and year date parts for interpreting **date**, **smalldatetime**, **datetime**, **datetime2**, and **datetimeoffset** character strings. DATEFORMAT is optional. If it isn't specified, the setting is inferred from the specified language.

For more information, see [SET DATEFORMAT \(Transact-SQL\)](#).

DELAYED_DURABILITY = { OFF | ON }

Applies to: SQL Server 2014 (12.x) and later versions, Azure SQL Database, SQL database in Microsoft Fabric Preview.

SQL Server transaction commits can be either fully durable, the default, or delayed durable.

For more information, see [Control Transaction Durability](#).

Simple examples

To help you get started, here are two quick examples: `SELECT DB_NAME() AS ThisDB;` returns the name of the current database. You can wrap that statement in a stored procedure, such as:

SQL

```
CREATE PROC What_DB_is_this
AS
SELECT DB_NAME() AS ThisDB;
```

Call the store procedure with statement: `EXEC What_DB_is_this;`

Slightly more complex, is to provide an input parameter to make the procedure more flexible. For example:

SQL

```
CREATE PROC What_DB_is_that @ID INT
AS
SELECT DB_NAME(@ID) AS ThatDB;
```

Provide a database ID number when you call the procedure. For example, `EXEC What_DB_is_that 2;` returns `tempdb`.

See [Examples](#) towards the end of this article for many more examples.

Best practices

Although this isn't an exhaustive list of best practices, these suggestions may improve procedure performance.

- Use the `SET NOCOUNT ON` statement as the first statement in the body of the procedure. That is, place it just after the `AS` keyword. This turns off messages that SQL Server sends back to the client after any `SELECT`, `INSERT`, `UPDATE`, `MERGE`, and `DELETE` statements are executed. This keeps the output generated to a minimum for clarity. There is no measurable performance benefit however on today's hardware. For information, see [SET NOCOUNT \(Transact-SQL\)](#).
- Use schema names when creating or referencing database objects in the procedure. It takes less processing time for the Database Engine to resolve object names if it doesn't have to search multiple schemas. It also prevents permission and access problems caused by a user's default schema being assigned when objects are created without specifying the schema.

- Avoid wrapping functions around columns specified in the WHERE and JOIN clauses. Doing so makes the columns non-deterministic and prevents the query processor from using indexes.
- Avoid using scalar functions in SELECT statements that return many rows of data. Because the scalar function must be applied to every row, the resulting behavior is like row-based processing and degrades performance.
- Avoid the use of `SELECT *`. Instead, specify the required column names. This can prevent some Database Engine errors that stop procedure execution. For example, a `SELECT *` statement that returns data from a 12 column table and then inserts that data into a 12 column temporary table succeeds until the number or order of columns in either table is changed.
- Avoid processing or returning too much data. Narrow the results as early as possible in the procedure code so that any subsequent operations performed by the procedure are done using the smallest data set possible. Send just the essential data to the client application. It is more efficient than sending extra data across the network and forcing the client application to work through unnecessarily large result sets.
- Use explicit transactions by using `BEGIN/COMMIT TRANSACTION` and keep transactions as short as possible. Longer transactions mean longer record locking and a greater potential for deadlocking.
- Use the Transact-SQL `TRY...CATCH` feature for error handling inside a procedure. `TRY...CATCH` can encapsulate an entire block of Transact-SQL statements. This not only creates less performance overhead, it also makes error reporting more accurate with significantly less programming.
- Use the `DEFAULT` keyword on all table columns that are referenced by `CREATE TABLE` or `ALTER TABLE` Transact-SQL statements in the body of the procedure. This prevents passing `NULL` to columns that don't allow null values.
- Use `NULL` or `NOT NULL` for each column in a temporary table. The `ANSI_DFLT_ON` and `ANSI_DFLT_OFF` options control the way the Database Engine assigns the `NULL` or `NOT NULL` attributes to columns when these attributes aren't specified in a `CREATE TABLE` or `ALTER TABLE` statement. If a connection executes a procedure with different settings for these options than the connection that created the procedure, the columns of the table created for the second connection can have different nullability and exhibit different behavior. If `NULL` or `NOT NULL` is explicitly stated for each column, the temporary tables are created by using the same nullability for all connections that execute the procedure.
- Use modification statements that convert nulls and include logic that eliminates rows with null values from queries. Be aware that in Transact-SQL, `NULL` isn't an empty or "nothing" value. It is a placeholder for an unknown value and can cause unexpected behavior, especially when querying for result sets or using `AGGREGATE` functions.

- Use the UNION ALL operator instead of the UNION or OR operators, unless there is a specific need for distinct values. The UNION ALL operator requires less processing overhead because duplicates aren't filtered out of the result set.

Remarks

There is no predefined maximum size of a procedure.

Variables specified in the procedure can be user-defined or system variables, such as @@SPID.

When a procedure is executed for the first time, it is compiled to determine an optimal access plan to retrieve the data. Subsequent executions of the procedure may reuse the plan already generated if it still remains in the plan cache of the Database Engine.

One or more procedures can execute automatically when SQL Server starts. The procedures must be created by the system administrator in the `master` database and executed under the **sysadmin** fixed server role as a background process. The procedures can't have any input or output parameters. For more information, see [Execute a Stored Procedure](#).

Procedures are nested when one procedure calls another or executes managed code by referencing a CLR routine, type, or aggregate. Procedures and managed code references can be nested up to 32 levels. The nesting level increases by one when the called procedure or managed code reference begins execution and decreases by one when the called procedure or managed code reference completes execution. Methods invoked from within the managed code don't count against the nesting level limit. However, when a CLR stored procedure performs data access operations through the SQL Server managed provider, an additional nesting level is added in the transition from managed code to SQL.

Attempting to exceed the maximum nesting level causes the entire calling chain to fail. You can use the @@NESTLEVEL function to return the nesting level of the current stored procedure execution.

Interoperability

The Database Engine saves the settings of both SET QUOTED_IDENTIFIER and SET ANSI_NULLS when a Transact-SQL procedure is created or modified. These original settings are used when the procedure is executed. Therefore, any client session settings for SET QUOTED_IDENTIFIER and SET ANSI_NULLS are ignored when the procedure is running.

Other SET options, such as SET ARITHABORT, SET ANSI_WARNINGS, or SET ANSI_PADDINGS aren't saved when a procedure is created or modified. If the logic of the procedure depends on a particular setting, include a SET statement at the start of the procedure to guarantee the appropriate setting. When a SET statement is executed from a procedure, the setting remains in effect only until the procedure has finished running. The setting is then restored to the value the procedure had when it was called. This enables individual clients to set the options they want without affecting the logic of the procedure.

Any SET statement can be specified inside a procedure, except SET SHOWPLAN_TEXT and SET SHOWPLAN_ALL. These must be the only statements in the batch. The SET option chosen remains in effect during the execution of the procedure and then reverts to its former setting.


 **Note**

SET ANSI_WARNINGS is not honored when passing parameters in a procedure, user-defined function, or when declaring and setting variables in a batch statement. For example, if a variable is defined as **char(3)**, and then set to a value larger than three characters, the data is truncated to the defined size and the INSERT or UPDATE statement succeeds.

Limitations and restrictions

The CREATE PROCEDURE statement can't be combined with other Transact-SQL statements in a single batch.

The following statements can't be used anywhere in the body of a stored procedure.

 Expand table

CREATE	SET	USE
CREATE AGGREGATE	SET SHOWPLAN_TEXT	USE <i>database_name</i>
CREATE DEFAULT	SET SHOWPLAN_XML	
CREATE RULE	SET PARSEONLY	
CREATE SCHEMA	SET SHOWPLAN_ALL	
CREATE or ALTER TRIGGER		
CREATE or ALTER FUNCTION		

CREATE	SET	USE
CREATE or ALTER PROCEDURE		
CREATE or ALTER VIEW		

A procedure can reference tables that don't yet exist. At creation time, only syntax checking is performed. The procedure isn't compiled until it is executed for the first time. Only during compilation are all objects referenced in the procedure resolved. Therefore, a syntactically correct procedure that references tables that don't exist can be created successfully; however, the procedure fails at execution time if the referenced tables don't exist.

You can't specify a function name as a parameter default value or as the value passed to a parameter when executing a procedure. However, you can pass a function as a variable as shown in the following example.

SQL

```
-- Passing the function value as a variable.
DECLARE @CheckDate DATETIME = GETDATE();
EXEC dbo.uspGetWhereUsedProductID 819, @CheckDate;
GO
```

If the procedure makes changes on a remote instance of SQL Server, the changes can't be rolled back. Remote procedures don't take part in transactions.

For the Database Engine to reference the correct method when it is overloaded in the .NET Framework, the method specified in the EXTERNAL NAME clause must have the following characteristics:

- Be declared as a static method.
- Receive the same number of parameters as the number of parameters of the procedure.
- Use parameter types that are compatible with the data types of the corresponding parameters of the SQL Server procedure. For information about matching SQL Server data types to the .NET Framework data types, see [Mapping CLR Parameter Data](#).

Metadata

The following table lists the catalog views and dynamic management views that you can use to return information about stored procedures.

[Expand table](#)

View	Description
sys.sql_modules	Returns the definition of a Transact-SQL procedure. The text of a procedure created with the ENCRYPTION option can't be viewed by using the <code>sys.sql_modules</code> catalog view.
sys.assembly_modules	Returns information about a CLR procedure.
sys.parameters	Returns information about the parameters that are defined in a procedure
sys.sql_expression_dependencies sys.dm_sql_referenced_entities sys.dm_sql_referencing_entities	Returns the objects that are referenced by a procedure.

To estimate the size of a compiled procedure, use the following Performance Monitor Counters.

[Expand table](#)

Performance Monitor object name	Performance Monitor Counter name
SQLServer: Plan Cache Object	Cache Hit Ratio
	Cache Pages
	Cache Object Counts ¹

¹ These counters are available for various categories of cache objects including ad hoc Transact-SQL, prepared Transact-SQL, procedures, triggers, and so on. For more information, see [SQL Server, Plan Cache Object](#).

Permissions

Requires `CREATE PROCEDURE` permission in the database and `ALTER` permission on the schema in which the procedure is being created, or requires membership in the **db_ddladmin** fixed database role.

For CLR stored procedures, requires ownership of the assembly referenced in the `EXTERNAL NAME` clause, or `REFERENCES` permission on that assembly.

CREATE PROCEDURE and memory-optimized tables

Memory-optimized tables can be accessed through both traditional and natively compiled stored procedures. Native procedures are in most cases the more efficient way. For more information, see [Natively Compiled Stored Procedures](#).

The following sample shows how to create a natively compiled stored procedure that accesses a memory-optimized table `dbo.Departments` :

SQL


```
CREATE PROCEDURE dbo.usp_add_kitchen @dept_id INT, @kitchen_count INT NOT NULL
WITH EXECUTE AS OWNER, SCHEMABINDING, NATIVE_COMPILATION
AS
BEGIN ATOMIC WITH (TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE = N'us_english')

UPDATE dbo.Departments
SET kitchen_count = ISNULL(kitchen_count, 0) + @kitchen_count
WHERE ID = @dept_id
END;
GO
```

A procedure created without `NATIVE_COMPILATION` can't be altered to a natively compiled stored procedure.

For a discussion of programmability in natively compiled stored procedures, supported query surface area, and operators see [Supported Features for Natively Compiled T-SQL Modules](#).

Examples

 Expand table

Category	Featured syntax elements
Basic Syntax	CREATE PROCEDURE
Passing parameters	<div>@parameter<ul style="list-style-type: none">= defaultOUTPUTtable-valued parameter typeCURSOR VARYING</div>

Category	Featured syntax elements
Modifying data by using a stored procedure	UPDATE
Error Handling	TRY...CATCH
Obfuscating the procedure definition	WITH ENCRYPTION
Forcing the Procedure to Recompile	WITH RECOMPILE
Setting the Security Context	EXECUTE AS

Basic syntax

Examples in this section demonstrate the basic functionality of the CREATE PROCEDURE statement using the minimum required syntax.

A. Create a Transact-SQL procedure

The following example creates a stored procedure that returns all employees (first and last names supplied), their job titles, and their department names from a view in the AdventureWorks2022 database. This procedure doesn't use any parameters. The example then demonstrates three methods of executing the procedure.

SQL

```
CREATE PROCEDURE HumanResources.uspGetAllEmployees
AS
    SET NOCOUNT ON;
    SELECT LastName, FirstName, JobTitle, Department
    FROM HumanResources.vEmployeeDepartment;
GO

SELECT * FROM HumanResources.vEmployeeDepartment;
```

The uspGetEmployees procedure can be executed in the following ways:

SQL

```
EXECUTE HumanResources.uspGetAllEmployees;
GO
-- Or
EXEC HumanResources.uspGetAllEmployees;
GO
```

```
-- Or, if this procedure is the first statement within a batch:
HumanResources.uspGetAllEmployees;
```

B. Return more than one result set

The following procedure returns two result sets.

SQL

```
CREATE PROCEDURE dbo.uspMultipleResults
AS
SELECT TOP(10) BusinessEntityID, Lastname, FirstName FROM Person.Person;
SELECT TOP(10) CustomerID, AccountNumber FROM Sales.Customer;
GO
```

C. Create a CLR stored procedure

The following example creates the `GetPhotoFromDB` procedure that references the `GetPhotoFromDB` method of the `LargeObjectBinary` class in the `HandlingLOBUsingCLR` assembly. Before the procedure is created, the `HandlingLOBUsingCLR` assembly is registered in the local database. The example assumes an assembly created from *assembly_bits*.

Applies to: SQL Server 2008 (10.0.x) and later versions, Azure SQL Database, SQL database in Microsoft Fabric Preview, when using an assembly created from *assembly_bits*.

SQL

```
CREATE ASSEMBLY HandlingLOBUsingCLR
FROM '\\MachineName\HandlingLOBUsingCLR\bin\Debug\HandlingLOBUsingCLR.dll';
GO
CREATE PROCEDURE dbo.GetPhotoFromDB
(
    @ProductPhotoID INT
    , @CurrentDirectory NVARCHAR(1024)
    , @FileName NVARCHAR(1024)
)
AS EXTERNAL NAME HandlingLOBUsingCLR.LargeObjectBinary.GetPhotoFromDB;
GO
```

Pass parameters

Examples in this section demonstrate how to use input and output parameters to pass values to and from a stored procedure.

D. Create a procedure with input parameters

The following example creates a stored procedure that returns information for a specific employee by passing values for the employee's first name and last name. This procedure accepts only exact matches for the parameters passed.

SQL

```
IF OBJECT_ID ( 'HumanResources.uspGetEmployees', 'P' ) IS NOT NULL
    DROP PROCEDURE HumanResources.uspGetEmployees;
GO
CREATE PROCEDURE HumanResources.uspGetEmployees
    @LastName NVARCHAR(50),
    @FirstName NVARCHAR(50)
AS

    SET NOCOUNT ON;
    SELECT FirstName, LastName, JobTitle, Department
    FROM HumanResources.vEmployeeDepartment
    WHERE FirstName = @FirstName AND LastName = @LastName;
GO
```

The `uspGetEmployees` procedure can be executed in the following ways:

SQL

```
EXECUTE HumanResources.uspGetEmployees N'Ackerman', N'Pilar';
-- Or
EXEC HumanResources.uspGetEmployees @LastName = N'Ackerman', @FirstName = N'Pilar';
GO
-- Or
EXECUTE HumanResources.uspGetEmployees @FirstName = N'Pilar', @LastName = N'Ackerman';
GO
-- Or, if this procedure is the first statement within a batch:
HumanResources.uspGetEmployees N'Ackerman', N'Pilar';
```

E. Use a procedure with wildcard parameters

The following example creates a stored procedure that returns information for employees by passing full or partial values for the employee's first name and last name. This procedure pattern

matches the parameters passed or, if not supplied, uses the preset default (last names that start with the letter D).

SQL

```
IF OBJECT_ID ( 'HumanResources.uspGetEmployees2', 'P' ) IS NOT NULL
    DROP PROCEDURE HumanResources.uspGetEmployees2;
GO
CREATE PROCEDURE HumanResources.uspGetEmployees2
    @LastName NVARCHAR(50) = N'D%',
    @FirstName NVARCHAR(50) = N '%'
AS
    SET NOCOUNT ON;
    SELECT FirstName, LastName, JobTitle, Department
    FROM HumanResources.vEmployeeDepartment
    WHERE FirstName LIKE @FirstName AND LastName LIKE @LastName;
```

The uspGetEmployees2 procedure can be executed in many combinations. Only a few possible combinations are shown here.

SQL

```
EXECUTE HumanResources.uspGetEmployees2;
-- Or
EXECUTE HumanResources.uspGetEmployees2 N'Wi%';
-- Or
EXECUTE HumanResources.uspGetEmployees2 @FirstName = N '%';
-- Or
EXECUTE HumanResources.uspGetEmployees2 N'[CK]ars[OE]n';
-- Or
EXECUTE HumanResources.uspGetEmployees2 N'Hesse', N'Stefen';
-- Or
EXECUTE HumanResources.uspGetEmployees2 N'H%', N'S%';
```

F. Use OUTPUT parameters

The following example creates the uspGetList procedure. This procedure returns a list of products that have prices that don't exceed a specified amount. The example shows using multiple SELECT statements and multiple OUTPUT parameters. OUTPUT parameters enable an external procedure, a batch, or more than one Transact-SQL statement to access a value set during the procedure execution.

SQL

```

IF OBJECT_ID ( 'Production.uspGetList', 'P' ) IS NOT NULL
    DROP PROCEDURE Production.uspGetList;
GO
CREATE PROCEDURE Production.uspGetList @Product VARCHAR(40)
    , @MaxPrice MONEY
    , @ComparePrice MONEY OUTPUT
    , @ListPrice MONEY OUT
AS
    SET NOCOUNT ON;
    SELECT p.[Name] AS Product, p.ListPrice AS 'List Price'
    FROM Production.Product AS p
    JOIN Production.ProductSubcategory AS s
        ON p.ProductSubcategoryID = s.ProductSubcategoryID
    WHERE s.[Name] LIKE @Product AND p.ListPrice < @MaxPrice;
-- Populate the output variable @ListPrice.
SET @ListPrice = (SELECT MAX(p.ListPrice)
    FROM Production.Product AS p
    JOIN Production.ProductSubcategory AS s
        ON p.ProductSubcategoryID = s.ProductSubcategoryID
    WHERE s.[Name] LIKE @Product AND p.ListPrice < @MaxPrice);
-- Populate the output variable @compareprice.
SET @ComparePrice = @MaxPrice;
GO

```

Execute `uspGetList` to return a list of Adventure Works products (Bikes) that cost less than \$700. The OUTPUT parameters `@Cost` and `@ComparePrices` are used with control-of-flow language to return a message in the **Messages** window.

ⓘ Note

The OUTPUT variable must be defined when the procedure is created and also when the variable is used. The parameter name and variable name do not have to match; however, the data type and parameter positioning must match, unless `@ListPrice = variable` is used.

SQL

```

DECLARE @ComparePrice MONEY, @Cost MONEY;
EXECUTE Production.uspGetList '%Bikes%', 700,
    @ComparePrice OUT,
    @Cost OUTPUT
IF @Cost <= @ComparePrice
BEGIN
    PRINT 'These products can be purchased for less than
    $'+RTRIM(CAST(@ComparePrice AS VARCHAR(20)))+'. '
END
ELSE

```

```
PRINT 'The prices for all products in this category exceed  
$'+ RTRIM(CAST(@ComparePrice AS VARCHAR(20)))+'. ';
```

Here is the partial result set:

Output

Product	List Price
-----	-----
Road-750 Black, 58	539.99
Mountain-500 Silver, 40	564.99
Mountain-500 Silver, 42	564.99
...	
Road-750 Black, 48	539.99
Road-750 Black, 52	539.99

(14 row(s) affected)

These items can be purchased for less than \$700.00.

G. Use a table-valued parameter

The following example uses a table-valued parameter type to insert multiple rows into a table. The example creates the parameter type, declares a table variable to reference it, fills the parameter list, and then passes the values to a stored procedure. The stored procedure uses the values to insert multiple rows into a table.

SQL

```
/* Create a table type. */  
CREATE TYPE LocationTableType AS TABLE  
( LocationName VARCHAR(50)  
, CostRate INT );  
GO  
  
/* Create a procedure to receive data for the table-valued parameter. */  
CREATE PROCEDURE usp_InsertProductionLocation  
    @TVP LocationTableType READONLY  
AS  
SET NOCOUNT ON  
INSERT INTO [AdventureWorks2022].[Production].[Location]  
    ([Name]  
    , [CostRate]  
    , [Availability]  
    , [ModifiedDate])  
SELECT *, 0, GETDATE()
```

```

FROM @TVP;
GO

/* Declare a variable that references the type. */
DECLARE @LocationTVP
AS LocationTableType;

/* Add data to the table variable. */
INSERT INTO @LocationTVP (LocationName, CostRate)
    SELECT [Name], 0.00
    FROM
        [AdventureWorks2022].[Person].[StateProvince];

/* Pass the table variable data to a stored procedure. */
EXEC usp_InsertProductionLocation @LocationTVP;
GO

```

H. Use an OUTPUT cursor parameter

The following example uses the OUTPUT cursor parameter to pass a cursor that is local to a procedure back to the calling batch, procedure, or trigger.

First, create the procedure that declares and then opens a cursor on the `Currency` table:

SQL

```

CREATE PROCEDURE dbo.uspCurrencyCursor
    @CurrencyCursor CURSOR VARYING OUTPUT
AS
    SET NOCOUNT ON;
    SET @CurrencyCursor = CURSOR
    FORWARD_ONLY STATIC FOR
        SELECT CurrencyCode, Name
        FROM Sales.Currency;
    OPEN @CurrencyCursor;
GO

```

Next, run a batch that declares a local cursor variable, executes the procedure to assign the cursor to the local variable, and then fetches the rows from the cursor.

SQL

```

DECLARE @MyCursor CURSOR;
EXEC dbo.uspCurrencyCursor @CurrencyCursor = @MyCursor OUTPUT;
WHILE (@@FETCH_STATUS = 0)
BEGIN;
    FETCH NEXT FROM @MyCursor;

```

```
END;  
CLOSE @MyCursor;  
DEALLOCATE @MyCursor;  
GO
```

Modify data by using a stored procedure

Examples in this section demonstrate how to insert or modify data in tables or views by including a Data Manipulation Language (DML) statement in the definition of the procedure.

I. Use UPDATE in a stored procedure

The following example uses an UPDATE statement in a stored procedure. The procedure takes one input parameter, @NewHours and one output parameter @RowCount. The @NewHours parameter value is used in the UPDATE statement to update the column vacationHours in the table HumanResources.Employee. The @RowCount output parameter is used to return the number of rows affected to a local variable. A CASE expression is used in the SET clause to conditionally determine the value that is set for vacationHours. When the employee is paid hourly (SalariedFlag = 0), vacationHours is set to the current number of hours plus the value specified in @NewHours; otherwise, vacationHours is set to the value specified in @NewHours.

SQL

```
CREATE PROCEDURE HumanResources.Update_VacationHours  
@NewHours SMALLINT, @Rowcount INT OUTPUT  
AS  
SET NOCOUNT ON;  
UPDATE HumanResources.Employee  
SET VacationHours =  
    ( CASE  
        WHEN SalariedFlag = 0 THEN VacationHours + @NewHours  
        ELSE @NewHours  
    END  
    )  
WHERE CurrentFlag = 1;  
SET @Rowcount = @@rowcount;  
  
GO  
DECLARE @Rowcount INT  
EXEC HumanResources.Update_VacationHours 40, @Rowcount OUTPUT  
PRINT @Rowcount;
```

Error handling

Examples in this section demonstrate methods to handle errors that might occur when the stored procedure is executed.

J. Use TRY...CATCH

The following example using the TRY...CATCH construct to return error information caught during the execution of a stored procedure.

SQL

```
CREATE PROCEDURE Production.uspDeleteWorkOrder ( @WorkOrderID INT )
AS
SET NOCOUNT ON;
BEGIN TRY
    BEGIN TRANSACTION
    -- Delete rows from the child table, WorkOrderRouting, for the specified work order.
    DELETE FROM Production.WorkOrderRouting
    WHERE WorkOrderID = @WorkOrderID;
    -- Delete the rows from the parent table, WorkOrder, for the specified work order.
    DELETE FROM Production.WorkOrder
    WHERE WorkOrderID = @WorkOrderID;
    COMMIT
END TRY

BEGIN CATCH
    -- Determine if an error occurred.
    IF @@TRANCOUNT > 0
        ROLLBACK

    -- Return the error information.
    DECLARE @ErrorMessage NVARCHAR(4000), @ErrorSeverity INT;
    SELECT @ErrorMessage = ERROR_MESSAGE(), @ErrorSeverity = ERROR_SEVERITY();
    RAISERROR(@ErrorMessage, @ErrorSeverity, 1);
END CATCH;

GO
EXEC Production.uspDeleteWorkOrder 13;
GO
/* Intentionally generate an error by reversing the order in which rows
   are deleted from the parent and child tables. This change does not
   cause an error when the procedure definition is altered, but produces
   an error when the procedure is executed.
*/
ALTER PROCEDURE Production.uspDeleteWorkOrder ( @WorkOrderID INT )
AS
```

```

BEGIN TRY
    BEGIN TRANSACTION
    -- Delete the rows from the parent table, WorkOrder, for the specified work order.
    DELETE FROM Production.WorkOrder
    WHERE WorkOrderID = @WorkOrderID;

    -- Delete rows from the child table, WorkOrderRouting, for the specified work order.
    DELETE FROM Production.WorkOrderRouting
    WHERE WorkOrderID = @WorkOrderID;
    COMMIT TRANSACTION
END TRY

BEGIN CATCH
    -- Determine if an error occurred.
    IF @@TRANCOUNT > 0
        ROLLBACK TRANSACTION

    -- Return the error information.
    DECLARE @ErrorMessage NVARCHAR(4000), @ErrorSeverity INT;
    SELECT @ErrorMessage = ERROR_MESSAGE(), @ErrorSeverity = ERROR_SEVERITY();
    RAISERROR(@ErrorMessage, @ErrorSeverity, 1);
END CATCH;
GO

-- Execute the altered procedure.
EXEC Production.uspDeleteWorkOrder 15;
GO
DROP PROCEDURE Production.uspDeleteWorkOrder;

```

Obfuscate the procedure definition

Examples in this section show how to obfuscate the definition of the stored procedure.

K. Use the WITH ENCRYPTION option

The following example creates the `HumanResources.uspEncryptThis` procedure.

Applies to: SQL Server 2008 (10.0.x) and later versions, Azure SQL Database, SQL database in Microsoft Fabric Preview.

SQL

```

CREATE PROCEDURE HumanResources.uspEncryptThis
WITH ENCRYPTION
AS
    SET NOCOUNT ON;
    SELECT BusinessEntityID, JobTitle, NationalIDNumber,
        VacationHours, SickLeaveHours

```

```
FROM HumanResources.Employee;  
GO
```

The `WITH ENCRYPTION` option obfuscates the definition of the procedure when querying the system catalog or using metadata functions, as shown by the following examples.

Run `sp_helptext`:

```
SQL  
  
EXEC sp_helptext 'HumanResources.uspEncryptThis';
```

Here's the result set.

The text for object `'HumanResources.uspEncryptThis'` is encrypted.

Directly query the `sys.sql_modules` catalog view:

```
SQL  
  
SELECT definition FROM sys.sql_modules  
WHERE object_id = OBJECT_ID('HumanResources.uspEncryptThis');
```

Here's the result set.

```
Output  
  
definition  
-----  
NULL
```

ⓘ Note

The system stored procedure `sp_helptext` is not supported in Azure Synapse Analytics. Instead, use the `sys.sql_modules` object catalog view.

Force the procedure to recompile

Examples in this section use the `WITH RECOMPILE` clause to force the procedure to recompile every time it is executed.

L. Use the WITH RECOMPILE option

The WITH RECOMPILE clause is helpful when the parameters supplied to the procedure aren't typical, and when a new execution plan shouldn't be cached or stored in memory.

SQL

```
IF OBJECT_ID ( 'dbo.uspProductByVendor', 'P' ) IS NOT NULL
    DROP PROCEDURE dbo.uspProductByVendor;
GO
CREATE PROCEDURE dbo.uspProductByVendor @Name VARCHAR(30) = '%'
WITH RECOMPILE
AS
    SET NOCOUNT ON;
    SELECT v.Name AS 'Vendor name', p.Name AS 'Product name'
    FROM Purchasing.Vendor AS v
    JOIN Purchasing.ProductVendor AS pv
        ON v.BusinessEntityID = pv.BusinessEntityID
    JOIN Production.Product AS p
        ON pv.ProductID = p.ProductID
    WHERE v.Name LIKE @Name;
```

Set the security context

Examples in this section use the EXECUTE AS clause to set the security context in which the stored procedure executes.

M. Use the EXECUTE AS clause

The following example shows using the EXECUTE AS clause to specify the security context in which a procedure can be executed. In the example, the option CALLER specifies that the procedure can be executed in the context of the user that calls it.

SQL

```
CREATE PROCEDURE Purchasing.uspVendorAllInfo
WITH EXECUTE AS CALLER
AS
    SET NOCOUNT ON;
    SELECT v.Name AS Vendor, p.Name AS 'Product name',
        v.CreditRating AS 'Rating',
        v.ActiveFlag AS Availability
    FROM Purchasing.Vendor v
    INNER JOIN Purchasing.ProductVendor pv
```

```
    ON v.BusinessEntityID = pv.BusinessEntityID
  INNER JOIN Production.Product p
    ON pv.ProductID = p.ProductID
  ORDER BY v.Name ASC;
GO
```

N. Create custom permission sets

The following example uses EXECUTE AS to create custom permissions for a database operation. Some operations such as TRUNCATE TABLE, don't have grantable permissions. By incorporating the TRUNCATE TABLE statement within a stored procedure and specifying that procedure execute as a user that has permissions to modify the table, you can extend the permissions to truncate the table to the user that you grant EXECUTE permissions on the procedure.

SQL

```
CREATE PROCEDURE dbo.TruncateMyTable
WITH EXECUTE AS SELF
AS TRUNCATE TABLE MyDB..MyTable;
```

Examples: Azure Synapse Analytics and Analytics Platform System (PDW)

O. Create a stored procedure that runs a SELECT statement

This example shows the basic syntax for creating and running a procedure. When running a batch, CREATE PROCEDURE must be the first statement. For example, to create the following stored procedure in **AdventureWorksPDW2022**, set the database context first, and then run the CREATE PROCEDURE statement.

SQL

```
-- Uses AdventureWorksDW database

--Run CREATE PROCEDURE as the first statement in a batch.
CREATE PROCEDURE Get10TopResellers
AS
BEGIN
    SELECT TOP (10) r.ResellerName, r.AnnualSales
    FROM DimReseller AS r
    ORDER BY AnnualSales DESC, ResellerName ASC;
```

```
END
```

```
;
```

```
GO
```

```
--Show 10 Top Resellers
```

```
EXEC Get10TopResellers;
```

See also

- [ALTER PROCEDURE \(Transact-SQL\)](#)
- [Control-of-Flow Language \(Transact-SQL\)](#)
- [Cursors](#)
- [Data Types \(Transact-SQL\)](#)
- [DECLARE @local_variable \(Transact-SQL\)](#)
- [DROP PROCEDURE \(Transact-SQL\)](#)
- [EXECUTE \(Transact-SQL\)](#)
- [EXECUTE AS \(Transact-SQL\)](#)
- [Stored Procedures \(Database Engine\)](#)
- [sp_procoption \(Transact-SQL\)](#)
- [sp_recompile \(Transact-SQL\)](#)
- [sys.sql_modules \(Transact-SQL\)](#)
- [sys.parameters \(Transact-SQL\)](#)
- [sys.procedures \(Transact-SQL\)](#)
- [sys.sql_expression_dependencies \(Transact-SQL\)](#)
- [sys.assembly_modules \(Transact-SQL\)](#)
- [sys.numbered_procedures \(Transact-SQL\)](#)
- [sys.numbered_procedure_parameters \(Transact-SQL\)](#)
- [OBJECT_DEFINITION \(Transact-SQL\)](#)
- [Create a Stored Procedure](#)
- [Use Table-Valued Parameters \(Database Engine\)](#)
- [sys.dm_sql_referenced_entities \(Transact-SQL\)](#)
- [sys.dm_sql_referencing_entities \(Transact-SQL\)](#)