Transact-SQL Data Definition Language (DDL) Reference

SQL Server 2012 Books Online

Reference

Microsoft

Transact-SQL Data Definition Language (DDL) Reference

SQL Server 2012 Books Online

Summary: Data Definition Language (DDL) is a vocabulary used to define data structures in SQL Server 2012. Use these statements to create, alter, or drop data structures in an instance of SQL Server.

Category: Reference
Applies to: SQL Server 2012
Source: SQL Server Books Online (link to source content)
E-book publication date: June 2012

Contents

Data Definition Language (DDL) Statements (Transact-SQL)

Data Definition Language (DDL) is a vocabulary used to define data structures in SQL Server 2012. Use these statements to create, alter, or drop data structures in an instance of SQL Server.

In this Section

ALTER Statements (Transact-SQL)
CREATE Statements (Transact-SQL)
DISABLE TRIGGER (Transact-SQL)
DISABLE TRIGGER (Transact-SQL)
DROP Statements (Transact-SQL)
ENABLE TRIGGER (Transact-SQL)
TRUNCATE TABLE (Transact-SQL)
UPDATE STATISTICS (Transact-SQL)

ALTER Statements

SQL Server Transact-SQL contains the following ALTER statements. Use ALTER statements to modify the definition of existing entities. For example, use ALTER TABLE to add a new column to a table, or use ALTER DATABASE to set database options.

In this Section

ALTER APPLICATION ROLE	ALTER EVENT SESSION	ALTER ROLE
ALTER ASSEMBLY	ALTER FULLTEXT CATALOG	ALTER ROUTE
ALTER ASYMMETRIC KEY	ALTER FULLTEXT INDEX	ALTER SCHEMA
ALTER AUTHORIZATION	ALTER FULLTEXT STOPLIST	ALTER SEARCH PROPERTY LIST (Transact-SQL)
ALTER BROKER PRIORITY	ALTER FUNCTION	ALTER SEQUENCE (Transact-SQL)
ALTER CERTIFICATE	ALTER INDEX	ALTER SERVER AUDIT

ALTER CREDENTIAL	ALTER LOGIN	ALTER SERVER AUDIT SPECIFICATION
ALTER CRYPTOGRAPHIC PROVIDER	ALTER MASTER KEY	ALTER SERVICE
ALTER DATABASE	ALTER MESSAGE TYPE	ALTER SERVICE MASTER KEY
ALTER DATABASE AUDIT SPECIFICATION	ALTER PARTITION FUNCTION	ALTER SYMMETRIC KEY
ALTER DATABASE Compatibility Level	ALTER PARTITION SCHEME	ALTER TABLE
ALTER DATABASE Database Mirroring	ALTER PROCEDURE	ALTER TRIGGER
ALTER DATABASE ENCRYPTION KEY	ALTER QUEUE	ALTER USER
ALTER DATABASE File and Filegroup Options	ALTER REMOTE SERVICE BINDING	ALTER VIEW
ALTER DATABASE SET Options	ALTER RESOURCE GOVERNOR	ALTER WORKLOAD GROUP
ALTER ENDPOINT	ALTER RESOURCE POOL	ALTER XML SCHEMA COLLECTION

See Also

CREATE Statements (Transact-SQL)

DROP Statements

ALTER APPLICATION ROLE

Changes the name, password, or default schema of an application role.

 Transact-SQL Syntax Conventions

Syntax

```
ALTER APPLICATION ROLE application_role_name
      WITH <set_item> [ ,...n ]

<set_item> ::=
```

NAME = **new_application_role_name**

| PASSWORD = **'password'**

| DEFAULT_SCHEMA = **schema_name**

Arguments

application_role_name

Is the name of the application role to be modified.

NAME = new_application_role_name

Specifies the new name of the application role. This name must not already be used to refer to any principal in the database.

PASSWORD = 'password'

Specifies the password for the application role. password must meet the Windows password policy requirements of the computer that is running the instance of SQL Server. You should always use strong passwords.

DEFAULT_SCHEMA = schema_name

Specifies the first schema that will be searched by the server when it resolves the names of objects. schema_name can be a schema that does not exist in the database.

Remarks

If the new application role name already exists in the database, the statement will fail. When the name, password, or default schema of an application role is changed the ID associated with the role is not changed.

Important

Password expiration policy is not applied to application role passwords. For this reason, take extra care in selecting strong passwords. Applications that invoke application roles must store their passwords.

Application roles are visible in the sys.database_principals catalog view.

Caution

In SQL Server 2005 the behavior of schemas changed from the behavior in earlier versions of SQL Server. Code that assumes that schemas are equivalent to database users may not return correct results. Old catalog views, including sysobjects, should not be used in a database in which any of the following DDL statements has ever been used: CREATE SCHEMA, ALTER SCHEMA, DROP SCHEMA, CREATE USER, ALTER USER, DROP USER, CREATE ROLE, ALTER ROLE, DROP ROLE, CREATE APPROLE, ALTER APPROLE, DROP APPROLE, ALTER AUTHORIZATION. In a database in which any of these statements has ever been used, you must use the new catalog views. The new catalog views take into account the separation of principals and schemas that is introduced in SQL Server 2005. For more information about catalog views, see EVENTDATA (Transact-SQL)).

Permissions

Requires ALTER ANY APPLICATION ROLE permission on the database. To change the default schema, the user also needs ALTER permission on the application role. An application role can alter its own default schema, but not its name or password.

Examples

A. Changing the name of application role

The following example changes the name of the application role `weekly_receipts` to `receipts_ledger`.

```
USE AdventureWorks2012;
CREATE APPLICATION ROLE weekly_receipts
    WITH PASSWORD = '987Gbv8$76sPYY5m23' ,
    DEFAULT_SCHEMA = Sales;
GO
ALTER APPLICATION ROLE weekly_receipts
    WITH NAME = receipts_ledger;
GO
```

B. Changing the password of application role

The following example changes the password of the application role `receipts_ledger`.

```
ALTER APPLICATION ROLE receipts_ledger
    WITH PASSWORD = '897yUUbv867y$200nk2i';
GO
```

C. Changing the name, password, and default schema

The following example changes the name, password, and default schema of the application role `receipts_ledger` all at the same time.

```
ALTER APPLICATION ROLE receipts_ledger
    WITH NAME = weekly_ledger,
    PASSWORD = '897yUUbv77bsrEE00nk2i',
    DEFAULT_SCHEMA = Production;
GO
```

See Also

Application Roles

CREATE APPLICATION ROLE (Transact-SQL)

DROP APPLICATION ROLE (Transact-SQL)

EVENTDATA (Transact-SQL)

ALTER ASSEMBLY

Alters an assembly by modifying the SQL Server catalog properties of an assembly. ALTER ASSEMBLY refreshes it to the latest copy of the Microsoft .NET Framework modules that hold its implementation and adds or removes files associated with it. Assemblies are created by using CREATE ASSEMBLY.

 Transact-SQL Syntax Conventions

Syntax

```
ALTER ASSEMBLY assembly_name
   [ FROM <client_assembly_specifier> | <assembly_bits> ]
   [ WITH <assembly_option> [ ,...n ] ]
   [ DROP FILE { file_name [ ,...n ] | ALL } ]
   [ ADD FILE FROM
   {
         client_file_specifier [ AS file_name ]
    | file_bits AS file_name
   } [,...n ]
   ] [ ; ]
<client_assembly_specifier> :: =
     '\\computer_name\share-name\[path\]manifest_file_name'
 | '[local_path\]manifest_file_name'

<assembly_bits> :: =
   { varbinary_literal | varbinary_expression }

<assembly_option> :: =
   PERMISSION_SET = { SAFE | EXTERNAL_ACCESS | UNSAFE }
 | VISIBILITY = { ON | OFF }
 | UNCHECKED DATA
```

Arguments

assembly_name

Is the name of the assembly you want to modify. assembly_name must already exist in the database.

FROM <client_assembly_specifier> | <assembly_bits>

Updates an assembly to the latest copy of the .NET Framework modules that hold its implementation. This option can only be used if there are no associated files with the specified assembly.

<client_assembly_specifier> specifies the network or local location where the assembly being refreshed is located. The network location includes the computer name, the share name and a path within that share. manifest_file_name specifies the name of the file that contains the manifest of the assembly.

<assembly_bits> is the binary value for the assembly.

Separate ALTER ASSEMBLY statements must be issued for any dependent assemblies that also require updating.

Note

This option is not available in a contained database.

PERMISSION_SET = { SAFE | EXTERNAL_ACCESS | UNSAFE }

Specifies the .NET Framework code access permission set property of the assembly. For more information about this property, see EVENTDATA (Transact-SQL).

Note

The EXTERNAL_ACCESS and UNSAFE options are not available in a contained database.

VISIBILITY = { ON | OFF }

Indicates whether the assembly is visible for creating common language runtime (CLR) functions, stored procedures, triggers, user-defined types, and user-defined aggregate functions against it. If set to OFF, the assembly is intended to be called only by other assemblies. If there are existing CLR database objects already created against the assembly, the visibility of the assembly cannot be changed. Any assemblies referenced by assembly_name are uploaded as not visible by default.

UNCHECKED DATA

By default, ALTER ASSEMBLY fails if it must verify the consistency of individual table rows. This option allows postponing the checks until a later time by using DBCC CHECKTABLE. If specified, SQL Server executes the ALTER ASSEMBLY statement even if there are tables in the database that contain the following:

- Persisted computed columns that either directly or indirectly reference methods in the assembly, through Transact-SQL functions or methods.
- CHECK constraints that directly or indirectly reference methods in the assembly.
- Columns of a CLR user-defined type that depend on the assembly, and the type implements a **UserDefined** (non-**Native**) serialization format.
- Columns of a CLR user-defined type that reference views created by using WITH SCHEMABINDING.

If any CHECK constraints are present, they are disabled and marked untrusted. Any tables

containing columns depending on the assembly are marked as containing unchecked data until those tables are explicitly checked.

Only members of the **db_owner** and **db_ddlowner** fixed database roles can specify this option.

For more information, see Implementing Assemblies.

[DROP FILE { file_name[,...n] | ALL }]

Removes the file name associated with the assembly, or all files associated with the assembly, from the database. If used with ADD FILE that follows, DROP FILE executes first. This lets you to replace a file with the same file name.

Note

This option is not available in a contained database.

[ADD FILE FROM { client_file_specifier [AS file_name] | file_bitsAS file_name}

Uploads a file to be associated with the assembly, such as source code, debug files or other related information, into the server and made visible in the **sys.assembly_files** catalog view. client_file_specifier specifies the location from which to upload the file. file_bits can be used instead to specify the list of binary values that make up the file. file_name specifies the name under which the file should be stored in the instance of SQL Server. file_name must be specified if file_bits is specified, and is optional if client_file_specifier is specified. If file_name is not specified, the file_name part of client_file_specifier is used as file_name.

Note

This option is not available in a contained database.

Remarks

ALTER ASSEMBLY does not disrupt currently running sessions that are running code in the assembly being modified. Current sessions complete execution by using the unaltered bits of the assembly.

If the FROM clause is specified, ALTER ASSEMBLY updates the assembly with respect to the latest copies of the modules provided. Because there might be CLR functions, stored procedures, triggers, data types, and user-defined aggregate functions in the instance of SQL Server that are already defined against the assembly, the ALTER ASSEMBLY statement rebinds them to the latest implementation of the assembly. To accomplish this rebinding, the methods that map to CLR functions, stored procedures, and triggers must still exist in the modified assembly with the same signatures. The classes that implement CLR user-defined types and user-defined aggregate functions must still satisfy the requirements for being a user-defined type or aggregate.

Caution

If WITH UNCHECKED DATA is not specified, SQL Server tries to prevent ALTER ASSEMBLY from executing if the new assembly version affects existing data in tables, indexes, or other persistent sites. However, SQL Server does not guarantee that computed columns,

indexes, indexed views or expressions will be consistent with the underlying routines and types when the CLR assembly is updated. Use caution when you execute ALTER ASSEMBLY to make sure that there is not a mismatch between the result of an expression and a value based on that expression stored in the assembly.

ALTER ASSEMBLY changes the assembly version. The culture and public key token of the assembly remain the same.

ALTER ASSEMBLY statement cannot be used to change the following:

- The signatures of CLR functions, aggregate functions, stored procedures, and triggers in an instance of SQL Server that reference the assembly. ALTER ASSEMBLY fails when SQL Server cannot rebind .NET Framework database objects in SQL Server with the new version of the assembly.
- The signatures of methods in the assembly that are called from other assemblies.
- The list of assemblies that depend on the assembly, as referenced in the **DependentList** property of the assembly.
- The indexability of a method, unless there are no indexes or persisted computed columns depending on that method, either directly or indirectly.
- The **FillRow** method name attribute for CLR table-valued functions.
- The **Accumulate** and **Terminate** method signature for user-defined aggregates.
- System assemblies.
- Assembly ownership. Use ALTER AUTHORIZATION (Transact-SQL) instead.

Additionally, for assemblies that implement user-defined types, ALTER ASSEMBLY can be used for making only the following changes:

- Modifying public methods of the user-defined type class, as long as signatures or attributes are not changed.
- Adding new public methods.
- Modifying private methods in any way.

Fields contained within a native-serialized user-defined type, including data members or base classes, cannot be changed by using ALTER ASSEMBLY. All other changes are unsupported.

If ADD FILE FROM is not specified, ALTER ASSEMBLY drops any files associated with the assembly.

If ALTER ASSEMBLY is executed without the UNCHECKED data clause, checks are performed to verify that the new assembly version does not affect existing data in tables. Depending on the amount of data that needs to be checked, this may affect performance.

Permissions

Requires ALTER permission on the assembly. Additional requirements are as follows:

- To alter an assembly whose existing permission set is EXTERNAL_ACCESS, the SQL Server login must have EXTERNAL ACCESS permission on the server.

- To alter an assembly whose existing permission set is UNSAFE requires membership in the **sysadmin** fixed server role.
- To change the permission set of an assembly to EXTERNAL_ACCESS, the SQL Server login must have EXTERNAL ACCESS ASSEMBLY permission on the server.
- To change the permission set of an assembly to UNSAFE requires membership in the **sysadmin** fixed server role.
- Specifying WITH UNCHECKED DATA requires membership in the **sysadmin** fixed server role.

For more information about assembly permission sets, see Designing Assemblies.

Examples

A. Refreshing an assembly

The following example updates assembly `ComplexNumber` to the latest copy of the .NET Framework modules that hold its implementation.

Note
Assembly `ComplexNumber` can be created by running the UserDefinedDataType sample scripts. For more information, see User-Defined Type (UDT) Sample.

```
ALTER ASSEMBLY ComplexNumber
FROM 'C:\Program Files\Microsoft SQL
Server\90\Tools\Samples\1033\Engine\Programmability\CLR\UserDefinedDataType\C
S\ComplexNumber\obj\Debug\ComplexNumber.dll'
```

B. Adding a file to associate with an assembly

The following example uploads the source code file `Class1.cs` to be associated with assembly `MyClass`. This example assumes assembly `MyClass` is already created in the database.

```
ALTER ASSEMBLY MyClass
ADD FILE FROM 'C:\MyClassProject\Class1.cs';
```

C. Changing the permissions of an assembly

The following example changes the permission set of assembly `ComplexNumber` from SAFE to `EXTERNAL ACCESS`.

```
ALTER ASSEMBLY ComplexNumber WITH PERMISSION_SET = EXTERNAL_ACCESS
```

See Also

CREATE ASSEMBLY
DROP ASSEMBLY
EVENTDATA (Transact-SQL)

ALTER ASYMMETRIC KEY

Changes the properties of an asymmetric key.

Transact-SQL Syntax Conventions

Syntax

ALTER ASYMMETRIC KEY **Asym_Key_Name** <alter_option>

<alter_option> ::=
<password_change_option>
|
REMOVE PRIVATE KEY

<password_change_option> ::=
WITH PRIVATE KEY **(** <password_option> [**,** <password_option>] **)**

<password_option> ::=
ENCRYPTION BY PASSWORD **=** '**strongPassword**'
|
DECRYPTION BY PASSWORD **=** '**oldPassword**'

Arguments

Asym_Key_Name

Is the name by which the asymmetric key is known in the database.

REMOVE PRIVATE KEY

Removes the private key from the asymmetric key The public key is not removed.

WITH PRIVATE KEY

Changes the protection of the private key.

ENCRYPTION BY PASSWORD = 'stongPassword'

Specifies a new password for protecting the private key. password must meet the Windows password policy requirements of the computer that is running the instance of SQL Server. If this option is omitted, the private key will be encrypted by the database master key.

DECRYPTION BY PASSWORD = 'oldPassword'

Specifies the old password, with which the private key is currently protected. Is not required if the private key is encrypted with the database master key.

Remarks

If there is no database master key the ENCRYPTION BY PASSWORD option is required, and the operation will fail if no password is supplied. For information about how to create a database master key, see OPEN MASTER KEY (Transact-SQL).

You can use ALTER ASYMMETRIC KEY to change the protection of the private key by specifying PRIVATE KEY options as shown in the following table.

Change protection from	ENCRYPTION BY PASSWORD	DECRYPTION BY PASSWORD
Old password to new password	Required	Required
Password to master key	Omit	Required
Master key to password	Required	Omit

The database master key must be opened before it can be used to protect a private key. For more information, see OPEN MASTER KEY (Transact-SQL).

To change the ownership of an asymmetric key, use ALTER AUTHORIZATION.

Permissions

Requires CONTROL permission on the asymmetric key if the private key is being removed.

Examples

A. Changing the password of the private key

The following example changes the password used to protect the private key of asymmetric key `PacificSales09`. The new password will be `<enterStrongPasswordHere>`.

```
ALTER ASYMMETRIC KEY PacificSales09
    WITH PRIVATE KEY (
    DECRYPTION BY PASSWORD = '<oldPassword>',
    ENCRYPTION BY PASSWORD = '<enterStrongPasswordHere>');
GO
```

B. Removing the private key from an asymmetric key

The following example removes the private key from `PacificSales19`, leaving only the public key.

```
ALTER ASYMMETRIC KEY PacificSales19 REMOVE PRIVATE KEY;
GO
```

C. Removing password protection from a private key

The following example removes the password protection from a private key and protects it with the database master key.

```
OPEN MASTER KEY;
ALTER ASYMMETRIC KEY PacificSales09 WITH PRIVATE KEY (
    DECRYPTION BY PASSWORD = '<enterStrongPasswordHere>' );
GO
```

See Also

CREATE ASYMMETRIC KEY (Transact-SQL)
DROP ASYMMETRIC KEY (Transact-SQL)
SQL Server and Database Encryption Keys (Database Engine)
Encryption Hierarchy
CREATE MASTER KEY (Transact-SQL)
OPEN MASTER KEY (Transact-SQL)
Understanding Extensible Key Management (EKM)

ALTER AUTHORIZATION

Changes the ownership of a securable.

 Transact-SQL Syntax Conventions

Syntax

```
ALTER AUTHORIZATION
  ON [ <class_type>:: ] entity_name
  TO { SCHEMA OWNER | principal_name }
[;]

<class_type> ::=
  {
     OBJECT | ASSEMBLY | ASYMMETRIC KEY | CERTIFICATE
  | CONTRACT | TYPE | DATABASE | ENDPOINT | FULLTEXT CATALOG
  | FULLTEXT STOPLIST | MESSAGE TYPE | REMOTE SERVICE BINDING
  | ROLE | ROUTE | SCHEMA | SEARCH PROPERTY LIST | SERVER ROLE
  | SERVICE | SYMMETRIC KEY | XML SCHEMA COLLECTION
  }
```

Arguments

<class_type>

Is the securable class of the entity for which the owner is being changed. OBJECT is the default.

entity_name

Is the name of the entity.

principal_name

Is the name of the principal that will own the entity.

Remarks

ALTER AUTHORIZATION can be used to change the ownership of any entity that has an owner. Ownership of database-contained entities can be transferred to any database-level principal. Ownership of server-level entities can be transferred only to server-level principals.

Important

Beginning with SQL Server 2005, a user can own an OBJECT or TYPE that is contained by a schema owned by another database user. This is a change of behavior from earlier versions of SQL Server. For more information, see OBJECTPROPERTY (Transact-SQL) and TYPEPROPERTY (Transact-SQL).

Ownership of the following schema-contained entities of type "object" can be transferred: tables, views, functions, procedures, queues, and synonyms.

Ownership of the following entities cannot be transferred: linked servers, statistics, constraints, rules, defaults, triggers, Service Broker queues, credentials, partition functions, partition schemes, database master keys, service master key, and event notifications.

Ownership of members of the following securable classes cannot be transferred: server, login, user, application role, and column.

The SCHEMA OWNER option is only valid when you are transferring ownership of a schema-contained entity. SCHEMA OWNER will transfer ownership of the entity to the owner of the schema in which it resides. Only entities of class OBJECT, TYPE, or XML SCHEMA COLLECTION are schema-contained.

If the target entity is not a database and the entity is being transferred to a new owner, all permissions on the target will be dropped.

Caution

In SQL Server 2005, the behavior of schemas changed from the behavior in earlier versions of SQL Server. Code that assumes that schemas are equivalent to database users may not return correct results. Old catalog views, including sysobjects, should not be used in a database in which any of the following DDL statements has ever been used: CREATE SCHEMA, ALTER SCHEMA, DROP SCHEMA, CREATE USER, ALTER USER, DROP USER, CREATE ROLE, ALTER ROLE, DROP ROLE, CREATE APPROLE, ALTER APPROLE, DROP APPROLE, ALTER AUTHORIZATION. In a database in which any of these statements has ever been used, you must use the new catalog views. The new catalog views take into account the separation of principals and schemas that was introduced in SQL Server 2005. For more information about catalog views, see Catalog Views (Transact-SQL).

Also, note the following:

Important

The only reliable way to find the owner of a object is to query the **sys.objects** catalog view. The only reliable way to find the owner of a type is to use the TYPEPROPERTY function.

Special Cases and Conditions

The following table lists special cases, exceptions, and conditions that apply to altering authorization.

Class	Condition
DATABASE	Cannot change the owner of system databases master, model, tempdb, the resource database, or a database that is used as a distribution database. The principal must be a login. If the principal is a Windows login without a corresponding SQL Server login, the principal must have CONTROL SERVER permission and TAKE OWNERSHIP permission on the database. If the principal is a SQL Server login, the principal cannot be mapped to a certificate or asymmetric key. Dependent aliases will be mapped to the new database owner. The DBO SID will be updated in both the current database and in sys.databases.
OBJECT	Cannot change ownership of triggers, constraints, rules, defaults, statistics, system objects, queues, indexed views, or tables with indexed views.
SCHEMA	When ownership is transferred, permissions on schema-contained objects that do not have explicit owners will be dropped. Cannot change the owner of sys, dbo, or information_schema.
TYPE	Cannot change ownership of a TYPE that belongs to sys or information_schema.
CONTRACT, MESSAGE TYPE, or SERVICE	Cannot change ownership of system entities.
SYMMETRIC KEY	Cannot change ownership of global temporary keys.

Class	Condition
CERTIFICATE or ASYMMETRIC KEY	Cannot transfer ownership of these entities to a role or group.
ENDPOINT	The principal must be a login.

Permissions

Requires TAKE OWNERSHIP permission on the entity. If the new owner is not the user that is executing this statement, also requires either, 1) IMPERSONATE permission on the new owner if it is a user or login; or 2) if the new owner is a role, membership in the role, or ALTER permission on the role; or 3) if the new owner is an application role, ALTER permission on the application role.

Examples

A. Transfer ownership of a table

The following example transfers ownership of table `Sprockets` to user `MichikoOsada`. The table is located inside schema `Parts`.

```
ALTER AUTHORIZATION ON OBJECT::Parts.Sprockets TO MichikoOsada;
GO
```

The query could also look like the following:

```
ALTER AUTHORIZATION ON Parts.Sprockets TO MichikoOsada;
GO
```

B. Transfer ownership of a view to the schema owner

The following example transfers ownership the view `ProductionView06` to the owner of the schema that contains it. The view is located inside schema `Production`.

```
ALTER AUTHORIZATION ON OBJECT::Production.ProductionView06 TO SCHEMA OWNER;
GO
```

C. Transfer ownership of a schema to a user

The following example transfers ownership of the schema `SeattleProduction11` to user `SandraAlayo`.

```
ALTER AUTHORIZATION ON SCHEMA::SeattleProduction11 TO SandraAlayo;
GO
```

D. Transfer ownership of an endpoint to a SQL Server login

The following example transfers ownership of endpoint `CantabSalesServer1` to `JaePak`. Because the endpoint is a server-level securable, the endpoint can only be transferred to a server-level principal.

```
ALTER AUTHORIZATION ON ENDPOINT::CantabSalesServer1 TO JaePak;
GO
```

See Also

OBJECTPROPERTY (Transact-SQL)
TYPEPROPERTY (Transact-SQL)
EVENTDATA (Transact-SQL)

ALTER AVAILABILITY GROUP

Alters an existing AlwaysOn availability group in SQL Server 2012. Most ALTER AVAILABILITY GROUP arguments are supported only on the server instance that hosts the current primary replica. However the JOIN, FAILOVER, and FORCE_FAILOVER_ALLOW_DATA_LOSS arguments, which can be run only on an secondary replica.

 Transact-SQL Syntax Conventions

Syntax

```
ALTER AVAILABILITY GROUP group_name
 {
   SET ( <set_option_spec> )
  | ADD DATABASE database_name
  | REMOVE DATABASE database_name
  | ADD REPLICA ON <add_replica_spec>
  | MODIFY REPLICA ON <modify_replica_spec>
  | REMOVE REPLICA ON <server_instance>
  | JOIN
  | FAILOVER
  | FORCE_FAILOVER_ALLOW_DATA_LOSS
  | ADD LISTENER 'dns_name' ( <add_listener_option> )
  | MODIFY LISTENER 'dns_name' ( <modify_listener_option> )
  | RESTART LISTENER 'dns_name'
  | REMOVE LISTENER 'dns_name'
 }
[;]
```

<set_option_spec> ::=

```
   AUTOMATED_BACKUP_PREFERENCE = { PRIMARY | SECONDARY_ONLY| SECONDARY | NONE
}
  | FAILURE_CONDITION_LEVEL  = { 1 | 2 | 3 | 4 | 5 }
  | HEALTH_CHECK_TIMEOUT = milliseconds

<server_instance> ::=
{ 'system_name[\instance_name]' | 'FCI_network_name[\instance_name]' }

<add_replica_spec>::=
  <server_instance> WITH
    (
      ENDPOINT_URL = 'TCP://system-address:port',
      AVAILABILITY_MODE = { SYNCHRONOUS_COMMIT | ASYNCHRONOUS_COMMIT },
      FAILOVER_MODE = { AUTOMATIC | MANUAL }
      [ , <add_replica_option> [ ,...n ] ]
    )

  <add_replica_option>::=
      BACKUP_PRIORITY = n
    | SECONDARY_ROLE ( {
        ALLOW_CONNECTIONS = { NO | READ_ONLY | ALL }
      | READ_ONLY_ROUTING_URL = 'TCP://system-address:port'
        } )
    | PRIMARY_ROLE ( {
        ALLOW_CONNECTIONS = { READ_WRITE | ALL }
      | READ_ONLY_ROUTING_LIST = { ( '<server_instance>' [ ,...n ] ) | NONE }
        } )
    | SESSION_TIMEOUT = seconds


<modify_replica_spec>::=
  <server_instance> WITH
    (
      ENDPOINT_URL = 'TCP://system-address:port'
    | AVAILABILITY_MODE = { SYNCHRONOUS_COMMIT | ASYNCHRONOUS_COMMIT }
    | FAILOVER_MODE = { AUTOMATIC | MANUAL }
```

| BACKUP_PRIORITY = **n**
| SECONDARY_ROLE **(** {
ALLOW_CONNECTIONS = { NO | READ_ONLY | ALL }
| READ_ONLY_ROUTING_URL = 'TCP**://system-address:port**'
} **)**
| PRIMARY_ROLE **(** {
ALLOW_CONNECTIONS = { READ_WRITE | ALL }
| READ_ONLY_ROUTING_LIST = { **(** '<server_instance>' [**,**...**n**] **)** | NONE }
} **)**
| SESSION_TIMEOUT = **seconds**
)

<add_listener_option> ::=
{
WITH DHCP [ON **(** <network_subnet_option> **)**]
| WITH IP **(** { **(** <ip_address_option> **)** } [**,** ...**n**] **)** [**,** PORT = **listener_port**]
}

<network_subnet_option> ::=
'**four_part_ipv4_address**', '**four_part_ipv4_mask**'

<ip_address_option> ::=
{
'**four_part_ipv4_address**', '**four_part_ipv4_mask**'
| '**ipv6_address**'
}

<modify_listener_option>::=
{
ADD IP **(** <ip_address_option> **)**
| PORT = **listener_port**
}

Arguments

group_name

Specifies the name of the new availability group. group_name must be a valid SQL Server identifier, and it must be unique across all availability groups in the WSFC cluster.

AUTOMATED_BACKUP_PREFERENCE = { PRIMARY | SECONDARY_ONLY| SECONDARY | NONE }

Specifies a preference about how a backup job should evaluate the primary replica when choosing where to perform backups. You can script a given backup job to take the automated backup preference into account. It is important to understand that the preference is not enforced by SQL Server, so it has no impact on ad-hoc backups.

Supported only on the primary replica.

The values are as follows:

PRIMARY

Specifies that the backups should always occur on the primary replica. This option is useful if you need backup features, such as creating differential backups, that are not supported when backup is run on a secondary replica.

SECONDARY_ONLY

Specifies that backups should never be performed on the primary replica. If the primary replica is the only replica online, the backup should not occur.

SECONDARY

Specifies that backups should occur on a secondary replica except when the primary replica is the only replica online. In that case, the backup should occur on the primary replica. This is the default behavior.

NONE

Specifies that you prefer that backup jobs ignore the role of the availability replicas when choosing the replica to perform backups. Note backup jobs might evaluate other factors such as backup priority of each availability replica in combination with its operational state and connected state.

There is no enforcement of the AUTOMATED_BACKUP_PREFERENCE setting. The interpretation of this preference depends on the logic, if any, that you script into back jobs for the databases in a given availability group. For more information, see Backup on Secondary Replicas (AlwaysOn Availability Groups).

Note

To view the automated backup preference of an existing availability group, select the **automated_backup_preference** or **automated_backup_preference_desc** column of the sys.availability_groups catalog view.

FAILURE_CONDITION_LEVEL = { 1 | 2 | 3 | 4 | 5 }

Specifies what failure conditions will trigger an automatic failover for this availability group. FAILURE_CONDITION_LEVEL is set at the group level but is relevant only on availability

replicas that are configured for synchronous-commit availability mode (AVAILIBILITY_MODE = SYNCHRONOUS_COMMIT). Furthermore, failure conditions can trigger an automatic failover only if both the primary and secondary replicas are configured for automatic failover mode (FAILOVER_MODE = AUTOMATIC) and the secondary replica is currently synchronized with the primary replica.

Supported only on the primary replica.

The failure-condition levels (1–5) range from the least restrictive, level 1, to the most restrictive, level 5. A given condition level encompasses all of the less restrictive levels. Thus, the strictest condition level, 5, includes the four less restrictive condition levels (1-4), level 4 includes levels 1-3, and so forth. The following table describes the failure-condition that corresponds to each level.

Level	Failure Condition
1	Specifies that an automatic failover should be initiated when any of the following occurs: • The SQL Server service is down. • The lease of the availability group for connecting to the WSFC cluster expires because no ACK is received from the server instance.
2	Specifies that an automatic failover should be initiated when any of the following occurs: • The instance of SQL Server does not connect to cluster, and the user-specified HEALTH_CHECK_TIMEOUT threshold of the availability group is exceeded. • The availability replica is in failed state.
3	Specifies that an automatic failover should be initiated on critical SQL Server internal errors, such as orphaned spinlocks, serious write-access violations, or too much dumping. This is the default behavior.
4	Specifies that an automatic failover should be initiated on moderate SQL Server internal errors, such as a persistent out-of-

	memory condition in the SQL Server internal resource pool.
5	Specifies that an automatic failover should be initiated on any qualified failure conditions, including: • Exhaustion of SQL Engine worker-threads. • Detection of an unsolvable deadlock.

nNote

Lack of response by an instance of SQL Server to client requests is not relevant to availability groups.

The FAILURE_CONDITION_LEVEL and HEALTH_CHECK_TIMEOUT values, define a *flexible failover policy* for a given group. This flexible failover policy provides you with granular control over what conditions must cause an automatic failover. For more information, see Flexible Failover Policy for Automatic Failover of an Availability Group (SQL Server).

HEALTH_CHECK_TIMEOUT = milliseconds

Specifies the wait time (in milliseconds) for the sp_server_diagnostics system stored procedure to return server-health information before WSFC cluster assumes that the server instance is slow or hung. HEALTH_CHECK_TIMEOUT is set at the group level but is relevant only on availability replicas that are configured for synchronous-commit availability mode with automatic failover (AVAILIBILITY_MODE = SYNCHRONOUS_COMMIT). Furthermore, a health-check timeout can trigger an automatic failover only if both the primary and secondary replicas are configured for automatic failover mode (FAILOVER_MODE = AUTOMATIC) and the secondary replica is currently synchronized with the primary replica.

The default HEALTH_CHECK_TIMEOUT value is 30000 milliseconds (30 seconds). The minimum value is 15000 milliseconds (15 seconds), and the maximum value is 4294967295 milliseconds.

Supported only on the primary replica.

Important

sp_server_diagnostics does not perform health checks at the database level.

ADD DATABASE database_name

Specifies a list of one or more user databases that you want to add to the availability group. These databases must reside on the instance of SQL Server that hosts the current primary replica. You can specify multiple databases for an availability group, but each database can belong to only one availability group. For information about the type of databases that an availability group can support, see Prerequisites, Restrictions, and Recommendations for AlwaysOn Availability Groups (SQL Server). To find out

which local databases already belong to an availability group, see the **replica_id** column in the sys.databases catalog view.

Supported only on the primary replica.

Note

After you have created the availability group, you will need connect to each server instance that hosts a secondary replica and then prepare each secondary database and join it to the availability group. For more information, see Start Data Movement on an AlwaysOn Secondary Database (SQL Server).

REMOVE DATABASE database_name

Removes the specified primary database and the corresponding secondary databases from the availability group. Supported only on the primary replica.

For information about the recommended follow up after removing an availability database from an availability group, see Remove a Primary Database from an Availability Group (SQL Server).

ADD REPLICA ON

Specifies from one to four SQL server instances to host secondary replicas in an availability group. Each replica is specified by its server instance address followed by a WITH (...) clause.

Supported only on the primary replica.

You need to join every new secondary replica to the availability group. For more information, see the description of the JOIN option, later in this section.

<server_instance>

Specifies the address of the instance of SQL Server that is the host for an replica. The address format depends on whether the instance is the default instance or a named instance and whether it is a standalone instance or a failover cluster instance (FCI). The syntax is as follows:

{ 'system_name[\instance_name]' | 'FCI_network_name[\instance_name]' }

The components of this address are as follows:

system_name

Is the NetBIOS name of the computer system on which the target instance of SQL Server resides. This computer must be a WSFC node.

FCI_network_name

Is the network name that is used to access a SQL Server failover cluster. Use this if the server instance participates as a SQL Server failover partner. Executing SELECT @@SERVERNAME on an FCI server instance returns its entire 'FCI_network_name[\instance_name]' string (which is the full replica name).

instance_name

Is the name of an instance of a SQL Server that is hosted by system_name or FCI_network_name and that has HADR service is enabled. For a default server instance,

instance_name is optional. The instance name is case insensitive. On a stand-alone server instance, this value name is the same as the value returned by executing SELECT @@SERVERNAME.

****

Is a separator used only when specifying instance_name, in order to separate it from system_name or FCI_network_name.

For information about the prerequisites for WSFC nodes and server instances, see Prerequisites, Restrictions, and Recommendations for AlwaysOn Availability Groups (SQL Server).

ENDPOINT_URL = 'TCP://system-address:port'

Specifies the URL path for the database mirroring endpoint on the instance of SQL Server that will host the availability replica that you are adding or modifying.

ENDPOINT_URL is required in the ADD REPLICA ON clause and optional in the MODIFY REPLICA ON clause. For more information, see Specify the Endpoint URL When Adding or Modifying an Availability Replica.

'TCP://system-address:port'

Specifies a URL for specifying an endpoint URL or read-only routing URL. The URL parameters are as follows:

system-address

Is a string, such as a system name, a fully qualified domain name, or an IP address, that unambiguously identifies the destination computer system.

port

Is a port number that is associated with the mirroring endpoint of the server instance (for the ENDPOINT_URL option) or the port number used by the Database Engine of the server instance (for the READ_ONLY_ROUTING_URL option).

AVAILABILITY_MODE = { SYNCHRONOUS_COMMIT | ASYNCHRONOUS_COMMIT }

Specifies whether the primary replica has to wait for the secondary replica to acknowledge the hardening (writing) of the log records to disk before the primary replica can commit the transaction on a given primary database. The transactions on different databases on the same primary replica can commit independently.

SYNCHRONOUS_COMMIT

Specifies that the primary replica will wait to commit transactions until they have been hardened on this secondary replica (synchronous-commit mode). You can specify SYNCHRONOUS_COMMIT for up to three replicas, including the primary replica.

ASYNCHRONOUS_COMMIT

Specifies that the primary replica commits transactions without waiting for this secondary replica to harden the log (synchronous-commit availability mode). You can specify

ASYNCHRONOUS_COMMIT for up to five availability replicas, including the primary replica.

AVAILABILITY_MODE is required in the ADD REPLICA ON clause and optional in the MODIFY REPLICA ON clause. For more information, see Availability Modes (AlwaysOn Availability Groups).

FAILOVER_MODE = { AUTOMATIC | MANUAL }

Specifies the failover mode of the availability replica that you are defining.

AUTOMATIC

Enables automatic failover. AUTOMATIC is supported only if you also specify AVAILABILITY_MODE = SYNCHRONOUS_COMMIT. You can specify AUTOMATIC for two availability replicas, including the primary replica.

Note

SQL Server Failover Cluster Instances (FCIs) do not support automatic failover by availability groups, so any availability replica that is hosted by an FCI can only be configured for manual failover.

MANUAL

Enables manual failover or forced manual failover (*forced failover*) by the database administrator.

FAILOVER_MODE is required in the ADD REPLICA ON clause and optional in the MODIFY REPLICA ON clause. Two types of manual failover exist, manual failover without data loss and forced failover (with possible data loss), which are supported under different conditions. For more information, see Failover Modes (AlwaysOn Availability Groups).

BACKUP_PRIORITY = n

Specifies your priority for performing backups on this replica relative to the other replicas in the same availability group. The value is an integer in the range of 0..100. These values have the following meanings:

- 1..100 indicates that the availability replica could be chosen for performing backups. 1 indicates the lowest priority, and 100 indicates the highest priority. If BACKUP_PRIORITY = 1, the availability replica would be chosen for performing backups only if no higher priority availability replicas are currently available.
- 0 indicates that this availability replica will never be chosen for performing backups. This is useful, for example, for a remote availability replica to which you never want backups to fail over.

For more information, see Backup on Secondary Replicas (AlwaysOn Availability Groups).

SECONDARY_ROLE (...)

Specifies role-specific settings that will take effect if this availability replica currently owns the secondary role (that is, whenever it is a secondary replica). Within the parentheses, specify either or both secondary-role options. If you specify both, use a comma-separated list.

The secondary role options are as follows:

ALLOW_CONNECTIONS = { NO | READ_ONLY | ALL }

Specifies whether the databases of a given availability replica that is performing the secondary role (that is, is acting as a secondary replica) can accept connections from clients, one of:

NO

No user connections are allowed to secondary databases of this replica. They are not available for read access. This is the default behavior.

READ_ONLY

Only connections are allowed to the databases in the secondary replica where the Application Intent property is set to **ReadOnly**. For more information about this property, see Using Connection String Keywords with SQL Server Native Client.

ALL

All connections are allowed to the databases in the secondary replica for read-only access.

For more information, see Read-Only Access to Secondary Replicas.

READ_ONLY_ROUTING_URL = 'TCP://system-address:port'

Specifies the URL to be used for routing read-intent connection requests to this availability replica. This is the URL on which the SQL Server Database Engine listens. Typically, the default instance of the SQL Server Database Engine listens on TCP port 1433.

For a named instance, you can obtain the port number by querying the **port** and **type_desc** columns of the sys.dm_tcp_listener_states dynamic management view. The server instance uses the Transact-SQL listener (**type_desc = 'TSQL'**).

Note

For a named instance of SQL Server, the Transact-SQL listener should be configured to use a specific port. For more information, see Configure a Server to Listen on a Specific TCP Port (SQL Server Configuration Manager).

PRIMARY_ROLE (...)

Specifies role-specific settings that will take effect if this availability replica currently owns the primary role (that is, whenever it is the primary replica). Within the parentheses, specify either or both primary-role options. If you specify both, use a comma-separated list.

The primary role options are as follows:

ALLOW_CONNECTIONS = { READ_WRITE | ALL }

Specifies the type of connection that the databases of a given availability replica that is performing the primary role (that is, is acting as a primary replica) can accept from clients, one of:

READ_WRITE

Connections where the Application Intent connection property is set to **ReadOnly** are disallowed. When the Application Intent property is set to **ReadWrite** or the Application Intent connection property is not set, the connection is allowed. For more information about Application Intent connection property, see Using Connection String Keywords with SQL Server Native Client.

ALL

All connections are allowed to the databases in the primary replica. This is the default behavior.

READ_ONLY_ROUTING_LIST = { ('<server_instance>' [,...n]) | NONE }

Specifies a comma-separated list of server instances that host availability replicas for this availability group that meet the following requirements when running under the secondary role:

- Be configured to allow all connections or read-only connections (see the ALLOW_CONNECTIONS argument of the SECONDARY_ROLE option, above).
- Have their read-only routing URL defined (see the READ_ONLY_ROUTING_URL argument of the SECONDARY_ROLE option, above).

The READ_ONLY_ROUTING_LIST values are as follows:

<server_instance>

Specifies the address of the instance of SQL Server that is the host for an availability replica that is a readable secondary replica when running under the secondary role.

Use a comma-separated list to specify all of the server instances that might host a readable secondary replica. Read-only routing will follow the order in which server instances are specified in the list. If you include a replica's host server instance on the replica's read-only routing list, placing this server instance at the end of the list is typically a good practice, so that read-intent connections go to a secondary replica, if one is available.

NONE

Specifies that when this availability replica is the primary replica, read-only routing will not be supported. This is the default behavior. When used with MODIFY REPLICA ON, this value disables an existing list, if any.

SESSION_TIMEOUT = seconds

Specifies the session-timeout period in seconds. If you do not specify this option, by default, the time period is 10 seconds. The minimum value is 5 seconds.

Important

We recommend that you keep the time-out period at 10 seconds or greater.

For more information about the session-timeout period, see Overview of AlwaysOn Availability Groups (SQL Server).

MODIFY REPLICA ON

Modifies any of the replicas of the availability group. The list of replicas to be modified contains the server instance address and a WITH (...) clause for each replica.

Supported only on the primary replica.

REMOVE REPLICA ON

Removes the specified secondary replica from the availability group. The current primary replica cannot be removed from an availability group. On being removed, the replica stops receiving data. Its secondary databases are removed from the availability group and enter the RESTORING state.

Supported only on the primary replica.

Note

If you remove a replica while it is unavailable or failed, when it comes back online it will discover that it no longer belongs the availability group.

JOIN

Causes the local server instance to host a secondary replica in the specified availability group.

Supported only on a secondary replica that has not yet been joined to the availability group.

For more information, see Join a Secondary Replica to an Availability Group (SQL Server).

FAILOVER

Initiates a manual failover of the availability group without data loss to the secondary replica to which you are connected. The secondary replica will take over the primary role and recover its copy of each database and bring them online as the new primary databases. The former primary replica concurrently transitions to the secondary role, and its databases become secondary databases and are immediately suspended. Potentially, these roles can be switched back and forth by a series of failures.

Note

A failover command returns as soon as the target secondary replica has accepted the command. However, database recovery occurs asynchronously after the availability group has finished failing over.

Supported only on a synchronous-commit secondary replica that is currently synchronized with the primary replica. Note that for a secondary replica to be synchronized the primary replica must also be running in synchronous-commit mode.

For information about the limitations, prerequisites and recommendations for a performing a planned manual failover, see Perform a Planned Manual Failover of an Availability Group (SQL Server).

FORCE_FAILOVER_ALLOW_DATA_LOSS

Caution

Forcing service, which might involve some data loss, is strictly a disaster recovery method. Therefore, We strongly recommend that you force failover only if the primary replica is no longer running, you are willing to risk losing data, and you must restore service to the availability group immediately.

Forces failover of the availability group, with possible data loss, to the secondary replica to which you are connected. The secondary replica will take over the primary role and recover its copy of each database and bring them online as the new primary databases. On any remaining secondary replicas, every secondary database is suspended until manually resumed. When the former primary replica becomes available, it will switch to the secondary role, and its databases will become suspended secondary databases.

Supported only on a secondary replica.

Note

A failover command returns as soon as the target secondary replica has accepted the command. However, database recovery occurs asynchronously after the availability group has finished failing over.

For information about the limitations, prerequisites and recommendations for forcing failover and the effect of a forced failover on the former primary databases, see Perform a Forced Manual Failover of an Availability Group (SQL Server).

ADD LISTENER 'dns_name' (<add_listener_option>)

Defines a new availability group listener for this availability group. Supported only on the primary replica.

Important

- Before you create your first listener, we strongly recommend that you read Prerequisites, Restrictions, and Recommendations for AlwaysOn Client Connectivity (SQL Server).
- After you create a listener for a given availability group, we strongly recommend that you do the following:

dns_name

Specifies the DNS host name of the availability group listener. The DNS name of the listener must be unique in the domain and in NetBIOS.

dns_name is a string value. This name can contain only alphanumeric characters, dashes (-), and hyphens (_), in any order. DNS host names are case insensitive. The maximum length is 63 characters.

We recommend that you specify a meaningful string. For example, for an availability group named `AG1`, a meaningful DNS host name would be `ag1-listener`.

Important

NetBIOS recognizes only the first 15 chars in the dns_name. If you have two WSFC clusters that are

controlled by the same Active Directory and you try to create availability group listeners in both of clusters using names with more than 15 characters and an identical 15 character prefix, you will get an error reporting that the Virtual Network Name resource could not be brought online. For information about prefix naming rules for DNS names, see Assigning Domain Names.

<add_listener_option>

ADD LISTENER takes one of the following options:

WITH DHCP [ON { ('four_part_ipv4_address', 'four_part_ipv4_mask') }]

Specifies that the availability group listener will use the Dynamic Host Configuration Protocol (DHCP). Optionally, use the ON clause to identify the network on which this listener will be created. DHCP is limited to a single subnet that is used for every server instances that hosts an availability replica in the availability group.

Important

We do not recommend DHCP in production environment. If there is a down time and the DHCP IP lease expires, extra time is required to register the new DHCP network IP address that is associated with the listener DNS name and impact the client connectivity. However, DHCP is good for setting up your development and testing environment to verify basic functions of availability groups and for integration with your applications.

For example:

```
WITH DHCP ON ('10.120.19.0','255.255.254.0')
```

WITH IP ({ ('four_part_ipv4_address', 'four_part_ipv4_mask') | ('ipv6_address') } [, ...n]) [, PORT = listener_port]

Specifies that, instead of using DHCP, the availability group listener will use one or more static IP addresses. To create an availability group across multiple subnets, each subnet requires one static IP address in the listener configuration. For a given subnet, the static IP address can be either an IPv4 address or an IPv6 address. Contact your network administrator to get a static IP address for each subnet that will host an availability replica for the new availability group.

For example:

```
WITH IP ( ('10.120.19.155','255.255.254.0') )
```

four_part_ipv4_address

Specifies an IPv4 four-part address for an availability group listener. For example, `10.120.19.155`.

four_part_ipv4_mask

Specifies an IPv4 four-part mask for an availability group listener. For example, `255.255.254.0`.

ipv6_address

Specifies an IPv6 address for an availability group listener. For example, `2001::4898:23:1002:20f:1fff:feff:b3a3`.

PORT = listener_port

Specifies the port number—listener_port—to be used by an availability group listener that is specified by a WITH IP clause. PORT is optional.

The default port number, 1433, is supported. However, if you have security concerns, we recommend using a different port number.

For example: `WITH IP ( ('2001::4898:23:1002:20f:1fff:feff:b3a3') ) , PORT = 7777`

MODIFY LISTENER 'dns_name' (<modify_listener_option>)

Modifies an existing availability group listener for this availability group. Supported only on the primary replica.

<modify_listener_option>

MODIFY LISTENER takes one of the following options:

ADD IP { ('four_part_ipv4_address', 'four_part_ipv4_mask') | ('dns_nameipv6_address') }

Adds the specified IP address to the availability group listener specified by dns_name.

PORT = listener_port

See the description of this argument earlier in this section.

RESTART LISTENER 'dns_name'

Restarts the listener that is associated with the specified DNS name. Supported only on the primary replica.

REMOVE LISTENER 'dns_name'

Removes the listener that is associated with the specified DNS name. Supported only on the primary replica.

Prerequisites and Restrictions

For information about prerequisites and restrictions on availability replicas and on their host server instances and computers, see Prerequisites, Restrictions, and Recommendations for AlwaysOn Availability Groups (SQL Server).

For information about restrictions on the AVAILABILITY GROUP Transact-SQL statements, see Overview of "HADR" Transact-SQL Statements (SQL Server).

Security

Permissions

Requires ALTER AVAILABILITY GROUP permission on the availability group, CONTROL AVAILABILITY GROUP permission, ALTER ANY AVAILABILITY GROUP permission, or CONTROL SERVER permission.

Examples

- A. Joining a secondary replica to an availability group
- B. Forcing failover of an availability group

A. Joining a secondary replica to an availability group

The following example, joins a secondary replica to which you are connected to the `AccountsAG` availability group.

```
ALTER AVAILABILITY GROUP AccountsAG JOIN;

GO
```

B. Forcing failover of an availability group

The following example forces the `AccountsAG` availability group to fail over to the secondary replica to which you are connected.

```
ALTER AVAILABILITY GROUP AccountsAG FORCE_FAILOVER_ALLOW_DATA_LOSS;

GO
```

See Also

CREATE AVAILABILITY GROUP (Transact-SQL)
ALTER DATABASE SET HADR (Transact-SQL)
DROP AVAILABILITY GROUP (Transact-SQL)
sys.availability_replicas (Transact-SQL)
sys.availability_groups (Transact-SQL)
Troubleshooting AlwaysOn Availability Groups Configuration (SQL Server)
Overview of AlwaysOn Availability Groups (SQL Server)
Client Connectivity and Application Failover (AlwaysOn Availability Groups)

ALTER BROKER PRIORITY

Changes the properties of a Service Broker conversation priority.

 Transact-SQL Syntax Conventions

Syntax

```
ALTER BROKER PRIORITY ConversationPriorityName
FOR CONVERSATION
{ SET ( [ CONTRACT_NAME = {ContractName | ANY } ]
      [ [ , ] LOCAL_SERVICE_NAME = {LocalServiceName | ANY } ]
      [ [ , ] REMOTE_SERVICE_NAME = {'RemoteServiceName' | ANY } ]
      [ [ , ] PRIORITY_LEVEL = { PriorityValue | DEFAULT } ]
```

)

}

[;]

Arguments

ConversationPriorityName

Specifies the name of the conversation priority to be changed. The name must refer to a conversation priority in the current database.

SET

Specifies the criteria for determining if the conversation priority applies to a conversation. SET is required and must contain at least one criterion: CONTRACT_NAME, LOCAL_SERVICE_NAME, REMOTE_SERVICE_NAME, or PRIORITY_LEVEL.

CONTRACT_NAME = {ContractName | ANY}

Specifies the name of a contract to be used as a criterion for determining if the conversation priority applies to a conversation. ContractName is a Database Engine identifier, and must specify the name of a contract in the current database.

ContractName

Specifies that the conversation priority can be applied only to conversations where the BEGIN DIALOG statement that started the conversation specified ON CONTRACT ContractName.

ANY

Specifies that the conversation priority can be applied to any conversation, regardless of which contract it uses.

If CONTRACT_NAME is not specified, the contract property of the conversation priority is not changed.

LOCAL_SERVICE_NAME = {LocalServiceName | ANY}

Specifies the name of a service to be used as a criterion to determine if the conversation priority applies to a conversation endpoint.

LocalServiceName is a Database Engine identifier and must specify the name of a service in the current database.

LocalServiceName

Specifies that the conversation priority can be applied to the following:

- Any initiator conversation endpoint whose initiator service name matches LocalServiceName.
- Any target conversation endpoint whose target service name matches LocalServiceName.

ANY

- Specifies that the conversation priority can be applied to any conversation endpoint, regardless of the name of the local service used by the endpoint.

If LOCAL_SERVICE_NAME is not specified, the local service property of the conversation priority is not changed.

REMOTE_SERVICE_NAME = {'RemoteServiceName' | ANY}

Specifies the name of a service to be used as a criterion to determine if the conversation priority applies to a conversation endpoint.

RemoteServiceName is a literal of type **nvarchar(256)**. Service Broker uses a byte-by-byte comparison to match the RemoteServiceName string. The comparison is case-sensitive and does not consider the current collation. The target service can be in the current instance of the Database Engine, or a remote instance of the Database Engine.

'RemoteServiceName'

Specifies the conversation priority be assigned to the following:

- Any initiator conversation endpoint whose associated target service name matches RemoteServiceName.
- Any target conversation endpoint whose associated initiator service name matches RemoteServiceName.

ANY

Specifies that the conversation priority applies to any conversation endpoint, regardless of the name of the remote service associated with the endpoint.

If REMOTE_SERVICE_NAME is not specified, the remote service property of the conversation priority is not changed.

PRIORITY_LEVEL = { PriorityValue | DEFAULT }

Specifies the priority level to assign any conversation endpoint that use the contracts and services that are specified in the conversation priority. PriorityValue must be an integer literal from 1 (lowest priority) to 10 (highest priority).

If PRIORITY_LEVEL is not specified, the priority level property of the conversation priority is not changed.

Remarks

No properties that are changed by ALTER BROKER PRIORITY are applied to existing conversations. The existing conversations continue with the priority that was assigned when they were started.

For more information, see CREATE BROKER PRIORITY (Transact-SQL).

Permissions

Permission for creating a conversation priority defaults to members of the **db_ddladmin** or **db_owner** fixed database roles, and to the **sysadmin** fixed server role. Requires ALTER permission on the database.

Examples

A. Changing only the priority level of an existing conversation priority.

Changes the priority level, but does not change the contract, local service, or remote service properties.

```
ALTER BROKER PRIORITY SimpleContractDefaultPriority
    FOR CONVERSATION
    SET (PRIORITY_LEVEL = 3);
```

B. Changing all of the properties of an existing conversation priority.

Changes the priority level, contract, local service, and remote service properties.

```
ALTER BROKER PRIORITY SimpleContractPriority
    FOR CONVERSATION
    SET (CONTRACT_NAME = SimpleContractB,
         LOCAL_SERVICE_NAME = TargetServiceB,
         REMOTE_SERVICE_NAME = N'InitiatorServiceB',
         PRIORITY_LEVEL = 8);
```

See Also

CREATE BROKER PRIORITY (Transact-SQL)
DROP BROKER PRIORITY (Transact-SQL)
sys.conversation_priorities (Transact-SQL)

ALTER CERTIFICATE

Changes the private key used to encrypt a certificate, or adds one if none is present. Changes the availability of a certificate to Service Broker.

 Transact-SQL Syntax Conventions

Syntax

```
ALTER CERTIFICATE certificate_name
   REMOVE PRIVATE KEY
   |
   WITH PRIVATE KEY ( <private_key_spec> [ ,... ] )
   |
```

```
WITH ACTIVE FOR BEGIN_DIALOG = [ ON | OFF ]
```

<private_key_spec> ::=

```
FILE = 'path_to_private_key'
|
DECRYPTION BY PASSWORD = 'key_password'
|
ENCRYPTION BY PASSWORD = 'password'
```

Arguments

certificate_name

Is the unique name by which the certificate is known in database.

FILE = 'path_to_private_key'

Specifies the complete path, including file name, to the private key. This parameter can be a local path or a UNC path to a network location. This file will be accessed within the security context of the SQL Server service account. When you use this option, you must make sure that the service account has access to the specified file.

DECRYPTION BY PASSWORD = 'key_password'

Specifies the password that is required to decrypt the private key.

ENCRYPTION BY PASSWORD = 'password'

Specifies the password used to encrypt the private key of the certificate in the database. password must meet the Windows password policy requirements of the computer that is running the instance of SQL Server. For more information, see EVENTDATA (Transact-SQL).

REMOVE PRIVATE KEY

Specifies that the private key should no longer be maintained inside the database.

ACTIVE FOR BEGIN_DIALOG = { ON | OFF }

Makes the certificate available to the initiator of a Service Broker dialog conversation.

Remarks

The private key must correspond to the public key specified by certificate_name.

The DECRYPTION BY PASSWORD clause can be omitted if the password in the file is protected with a null password.

When the private key of a certificate that already exists in the database is imported from a file, the private key will be automatically protected by the database master key. To protect the private key with a password, use the ENCRYPTION BY PASSWORD phrase.

The REMOVE PRIVATE KEY option will delete the private key of the certificate from the database. You can do this when the certificate will be used to verify signatures or in Service Broker scenarios that do not require a private key. Do not remove the private key of a certificate that protects a symmetric key.

You do not have to specify a decryption password when the private key is encrypted by using the database master key.

Important
Always make an archival copy of a private key before removing it from a database. For more information, see BACKUP CERTIFICATE (Transact-SQL).

The WITH PRIVATE KEY option is not available in a contained database.

Permissions

Requires ALTER permission on the certificate.

Examples

A. Changing the password of a certificate

```
ALTER CERTIFICATE Shipping04
    WITH PRIVATE KEY (DECRYPTION BY PASSWORD = 'pGF$5DGvbd2439587y',
    ENCRYPTION BY PASSWORD = '4-329578thlkajdshglXCSgf');
GO
```

B. Changing the password that is used to encrypt the private key

```
ALTER CERTIFICATE Shipping11
    WITH PRIVATE KEY (ENCRYPTION BY PASSWORD = '34958tosdgfkh##38',
    DECRYPTION BY PASSWORD = '95hkjdskghFDGGG4%');
GO
```

C. Importing a private key for a certificate that is already present in the database

```
ALTER CERTIFICATE Shipping13
    WITH PRIVATE KEY (FILE = 'c:\\importedkeys\Shipping13',
    DECRYPTION BY PASSWORD = 'GDFLKl8^^GGG4000%');
GO
```

D. Changing the protection of the private key from a password to the database master key

```
ALTER CERTIFICATE Shipping15
    WITH PRIVATE KEY (DECRYPTION BY PASSWORD = '95hk000eEnvjkjy#F%');
GO
```

See Also

CREATE CERTIFICATE (Transact-SQL)
DROP CERTIFICATE (Transact-SQL)
BACKUP CERTIFICATE (Transact-SQL)
Encryption Hierarchy
EVENTDATA (Transact-SQL)

ALTER CREDENTIAL

Changes the properties of a credential.

 Transact-SQL Syntax Conventions

Syntax

```
ALTER CREDENTIAL credential_name WITH IDENTITY = 'identity_name'
   [ , SECRET = 'secret' ]
```

Arguments

credential_name

Specifies the name of the credential that is being altered.

IDENTITY = 'identity_name'

Specifies the name of the account to be used when connecting outside the server.

SECRET = 'secret'

Specifies the secret required for outgoing authentication. secret is optional.

Remarks

When a credential is changed, the values of both identity_name and secret are reset. If the optional SECRET argument is not specified, the value of the stored secret will be set to NULL.

The secret is encrypted by using the service master key. If the service master key is regenerated, the secret is reencrypted by using the new service master key.

Information about credentials is visible in the **sys.credentials** catalog view.

Permissions

Requires ALTER ANY CREDENTIAL permission. If the credential is a system credential, requires CONTROL SERVER permission.

Examples

A. Changing the password of a credential

The following example changes the secret stored in a credential called `Saddles`. The credential contains the Windows login `RettigB` and its password. The new password is added to the credential using the SECRET clause.

```
ALTER CREDENTIAL Saddles WITH IDENTITY = 'RettigB',
    SECRET = 'sdrlk8$40-dksli87nNN8';
GO
```

B. Removing the password from a credential

The following example removes the password from a credential named `Frames`. The credential contains Windows login `Aboulrus8` and a password. After the statement is executed, the credential will have a NULL password because the SECRET option is not specified.

```
ALTER CREDENTIAL Frames WITH IDENTITY = 'Aboulrus8';
GO
```

See Also

sys.credentials (Transact-SQL)
CREATE CREDENTIAL (Transact-SQL)
DROP CREDENTIAL (Transact-SQL)
CREATE LOGIN (Transact-SQL)
sys.credentials (Transact-SQL)

ALTER CRYPTOGRAPHIC PROVIDER

Alters a cryptographic provider within SQL Server from an Extensible Key Management (EKM) provider.

 Transact-SQL Syntax Conventions

Syntax

ALTER CRYPTOGRAPHIC PROVIDER **`provider_name`**
[FROM FILE = **`path_of_DLL`**]
ENABLE | DISABLE

Arguments

provider_name

Name of the Extensible Key Management provider.

Path_of_DLL

Path of the .dll file that implements the SQL Server Extensible Key Management interface.

ENABLE | DISABLE

Enables or disables a provider.

Remarks

If the provider changes the .dll file that is used to implement Extensible Key Management in SQL Server, you must use the ALTER CRYPTOGRAPHIC PROVIDER statement.

When the .dll file path is updated by using the ALTER CRYPTOGRAPHIC PROVIDER statement, SQL Server performs the following actions:

- Disables the provider.
- Verifies the DLL signature and ensures that the .dll file has the same GUID as the one recorded in the catalog.
- Updates the DLL version in the catalog.

When an EKM provider is set to DISABLE, any attempts on new connections to use the provider with encryption statements will fail.

To disable a provider, all sessions that use the provider must be terminated.

When an EKM provider dll does not implement all of the necessary methods, ALTER CRYPTOGRAPHIC PROVIDER can return error 33085:

```
One or more methods cannot be found in cryptographic provider library
'%.*ls'.
```

When the header file used to create the EKM provider dll is out of date, ALTER CRYPTOGRAPHIC PROVIDER can return error 33032:

```
SQL Crypto API version '%02d.%02d' implemented by provider is not supported.
Supported version is '%02d.%02d'.
```

Permissions

Requires CONTROL permission on the cryptographic provider.

Examples

The following example alters a cryptographic provider, called `SecurityProvider` in SQL Server, to a newer version of a .dll file. This new version is named `c:\SecurityProvider\SecurityProvider_v2.dll` and is installed on the server. The provider's certificate must be installed on the server.

```
/* First, disable the provider to perform the upgrade.

This will terminate all open cryptographic sessions */

ALTER CRYPTOGRAPHIC PROVIDER SecurityProvider

DISABLE;

GO


/* Upgrade the provider .dll file. The GUID must the same
```

```
as the previous version, but the version can be different. */
ALTER CRYPTOGRAPHIC PROVIDER SecurityProvider
FROM FILE = 'c:\SecurityProvider\SecurityProvider_v2.dll';
GO

/* Enable the upgraded provider. */
ALTER CRYPTOGRAPHIC PROVIDER SecurityProvider
ENABLE;
GO
```

See Also

Understanding Extensible Key Management (EKM)
CREATE CRYPTOGRAPHIC PROVIDER (Transact-SQL)
DROP CRYPTOGRAPHIC PROVIDER (Transact-SQL)
CREATE SYMMETRIC KEY (Transact-SQL)

ALTER DATABASE

Modifies a database, or the files and filegroups associated with the database. Adds or removes files and filegroups from a database, changes the attributes of a database or its files and filegroups, changes the database collation, and sets database options. Database snapshots cannot be modified. To modify database options associated with replication, use sp_replicationdboption.

Because of its length, the ALTER DATABASE syntax is separated into the following topics:

ALTER DATABASE

The current topic provides the syntax for changing the name and the collation of a database.

ALTER DATABASE File and Filegroup Options

Provides the syntax for adding and removing files and filegroups from a database, and for changing the attributes of the files and filegroups.

ALTER DATABASE SET Options

Provides the syntax for changing the attributes of a database by using the SET options of ALTER DATABASE.

ALTER DATABASE Database Mirroring

Provides the syntax for the SET options of ALTER DATABASE that are related to database mirroring.

ALTER DATABASE SET HADR

Provides the syntax for the AlwaysOn Availability Groups options of ALTER DATABASE for

configuring a secondary database on a secondary replica of an AlwaysOn availability group.

ALTER DATABASE Compatibility Level

Provides the syntax for the SET options of ALTER DATABASE that are related to database compatibility levels.

 Transact-SQL Syntax Conventions

Syntax

```
ALTER DATABASE { database_name  | CURRENT }
{
    MODIFY NAME = new_database_name
  | COLLATE collation_name
  | <file_and_filegroup_options>
  | <set_database_options>
}
[;]

<file_and_filegroup_options >::=
  <add_or_modify_files>::=
  <filespec>::=
  <add_or_modify_filegroups>::=
  <filegroup_updatability_option>::=

<set_database_options>::=
  <optionspec>::=
  <auto_option> ::=
  <change_tracking_option> ::=
  <cursor_option> ::=
  <database_mirroring_option> ::=
  <date_correlation_optimization_option> ::=
  <db_encryption_option> ::=
  <db_state_option> ::=
  <db_update_option> ::=
  <db_user_access_option> ::=
  <external_access_option> ::=
  <FILESTREAM_options> ::=
```

<HADR_options> ::=

<parameterization_option> ::=

<recovery_option> ::=

<service_broker_option> ::=

<snapshot_option> ::=

<sql_option> ::=

<termination> ::=

Arguments

database_name

Is the name of the database to be modified.

Note

This option is not available in a Contained Database.

CURRENT

Designates that the current database in use should be altered.

CONTAINMENT

Specifies the containment status of the database. OFF = non-contained database. PARTIAL = partially contained database.

MODIFY NAME = new_database_name

Renames the database with the name specified as new_database_name.

COLLATE collation_name

Specifies the collation for the database. collation_name can be either a Windows collation name or a SQL collation name. If not specified, the database is assigned the collation of the instance of SQL Server.

For more information about the Windows and SQL collation names, see COLLATE (Transact-SQL).

<file_and_filegroup_options >::=

For more information, see ALTER DATABASE File and Filegroup Options (Transact-SQL).

<set_database_options >::=

For more information, see ALTER DATABASE SET Options (Transact-SQL), ALTER DATABASE Database Mirroring (Transact-SQL), ALTER DATABASE SET HADR (Transact-SQL), and ALTER DATABASE Compatibility Level (Transact-SQL).

Remarks

To remove a database, use DROP DATABASE.

To decrease the size of a database, use DBCC SHRINKDATABASE.

The ALTER DATABASE statement must run in autocommit mode (the default transaction management mode) and is not allowed in an explicit or implicit transaction.

In SQL Server 2005 or later, the state of a database file (for example, online or offline), is maintained independently from the state of the database. For more information, see File States. The state of the files within a filegroup determines the availability of the whole filegroup. For a filegroup to be available, all files within the filegroup must be online. If a filegroup is offline, any try to access the filegroup by an SQL statement will fail with an error. When you build query plans for SELECT statements, the query optimizer avoids nonclustered indexes and indexed views that reside in offline filegroups. This enables these statements to succeed. However, if the offline filegroup contains the heap or clustered index of the target table, the SELECT statements fail. Additionally, any INSERT, UPDATE, or DELETE statement that modifies a table with any index in an offline filegroup will fail.

When a database is in the RESTORING state, most ALTER DATABASE statements will fail. The exception is setting database mirroring options. A database may be in the RESTORING state during an active restore operation or when a restore operation of a database or log file fails because of a corrupted backup file.

The plan cache for the instance of SQL Server is cleared by setting one of the following options:

OFFLINE	READ_WRITE
ONLINE	MODIFY FILEGROUP DEFAULT
MODIFY_NAME	MODIFY FILEGROUP READ_WRITE
COLLATE	MODIFY FILEGROUP READ_ONLY
READ_ONLY	

Clearing the plan cache causes a recompilation of all subsequent execution plans and can cause a sudden, temporary decrease in query performance. For each cleared cachestore in the plan cache, the SQL Server error log contains the following informational message: "SQL Server has encountered %d occurrence(s) of cachestore flush for the '%s' cachestore (part of plan cache) due to some database maintenance or reconfigure operations". This message is logged every five minutes as long as the cache is flushed within that time interval.

Changing the Database Collation

Before you apply a different collation to a database, make sure that the following conditions are in place:

1. You are the only one currently using the database.
2. No schema-bound object depends on the collation of the database.

If the following objects, which depend on the database collation, exist in the database, the ALTER DATABASE database_name COLLATE statement will fail. SQL Server will return an error message for each object blocking the ALTER action:

- User-defined functions and views created with SCHEMABINDING.
- Computed columns.
- CHECK constraints.
- Table-valued functions that return tables with character columns with collations inherited from the default database collation.

Dependency information for non-schema-bound entities is automatically updated when the database collation is changed.

Changing the database collation does not create duplicates among any system names for the database objects. If duplicate names result from the changed collation, the following namespaces may cause the failure of a database collation change:

- Object names such as a procedure, table, trigger, or view.
- Schema names
- Principals such as a group, role, or user.
- Scalar-type names such as system and user-defined types.
- Full-text catalog names.
- Column or parameter names within an object.
- Index names within a table.

Duplicate names resulting from the new collation will cause the change action to fail, and SQL Server will return an error message specifying the namespace where the duplicate was found.

Viewing Database Information

You can use catalog views, system functions, and system stored procedures to return information about databases, files, and filegroups.

Permissions

Requires ALTER permission on the database.

Examples

A. Changing the name of a database

The following example changes the name of the `AdventureWorks2012` database to `Northwind`.

```
USE master;
GO
ALTER DATABASE AdventureWorks2012
Modify Name = Northwind ;
GO
```

B. Changing the collation of a database

The following example creates a database named `testdb` with the `SQL_Latin1_General_CP1_CI_AS` collation, and then changes the collation of the `testdb` database to `COLLATE French_CI_AI`.

```
USE master;
GO

CREATE DATABASE testdb
COLLATE SQL_Latin1_General_CP1_CI_AS ;
GO

ALTER DATABASE testDB
COLLATE French_CI_AI ;
GO
```

See Also

CREATE DATABASE
DATABASEPROPERTYEX
DROP DATABASE
SET TRANSACTION ISOLATION LEVEL
EVENTDATA
sp_configure
sp_spaceused
sys.databases (Transact-SQL)
sys.database_files
sys.database_mirroring_witnesses
sys.data_spaces (Transact-SQL)
sys.filegroups
sys.master_files (Transact-SQL)
System Databases

ALTER DATABASE File and Filegroup Options

Modifies the files and filegroups associated with the database. Adds or removes files and filegroups from a database, and changes the attributes of a database or its files and filegroups. For other ALTER DATABASE options, see ALTER DATABASE (Transact-SQL).

 Transact-SQL Syntax Conventions

Syntax

```
ALTER DATABASE database_name
{
    <add_or_modify_files>
  | <add_or_modify_filegroups>
}
[;]

<add_or_modify_files>::=
{
    ADD FILE <filespec> [ ,...n ]
        [ TO FILEGROUP { filegroup_name } ]
  | ADD LOG FILE <filespec> [ ,...n ]
  | REMOVE FILE logical_file_name
  | MODIFY FILE <filespec>
}

<filespec>::=
(
    NAME = logical_file_name
    [ , NEWNAME = new_logical_name ]
    [ , FILENAME = {'os_file_name' | 'filestream_path' } ]
    [ , SIZE = size [ KB | MB | GB | TB ] ]
    [ , MAXSIZE = { max_size [ KB | MB | GB | TB ] | UNLIMITED } ]
    [ , FILEGROWTH = growth_increment [ KB | MB | GB | TB| % ] ]
    [ , OFFLINE ]
)

<add_or_modify_filegroups>::=
{
    | ADD FILEGROUP filegroup_name
        [ CONTAINS FILESTREAM ]
    | REMOVE FILEGROUP filegroup_name
    | MODIFY FILEGROUP filegroup_name
```

{ <filegroup_updatability_option>
| DEFAULT
| NAME = **new_filegroup_name**
}
}

<filegroup_updatability_option>::=

{
{ READONLY | READWRITE }
| { READ_ONLY | READ_WRITE }
}

Arguments

<add_or_modify_files>::=

Specifies the file to be added, removed, or modified.

database_name

Is the name of the database to be modified.

ADD FILE

Adds a file to the database.

TO FILEGROUP { filegroup_name }

Specifies the filegroup to which to add the specified file. To display the current filegroups and which filegroup is the current default, use the sys.filegroups catalog view.

ADD LOG FILE

Adds a log file be added to the specified database.

REMOVE FILE logical_file_name

Removes the logical file description from an instance of SQL Server and deletes the physical file. The file cannot be removed unless it is empty.

logical_file_name

Is the logical name used in SQL Server when referencing the file.

MODIFY FILE

Specifies the file that should be modified. Only one <filespec> property can be changed at a time. NAME must always be specified in the <filespec> to identify the file to be modified. If SIZE is specified, the new size must be larger than the current file size.

To modify the logical name of a data file or log file, specify the logical file name to be renamed in the NAME clause, and specify the new logical name for the file in the NEWNAME clause. For example:

```
MODIFY FILE ( NAME = logical_file_name, NEWNAME =
```

new_logical_name)

To move a data file or log file to a new location, specify the current logical file name in the NAME clause and specify the new path and operating system file name in the FILENAME clause. For example:

```
MODIFY FILE ( NAME = logical_file_name, FILENAME = '
new_path/os_file_name ' )
```

When you move a full-text catalog, specify only the new path in the FILENAME clause. Do not specify the operating-system file name.

For more information, see Moving Database Files.

For a FILESTREAM filegroup, NAME can be modified online. FILENAME can be modified online; however, the change does not take effect until after the container is physically relocated and the server is shutdown and then restarted.

You can set a FILESTREAM file to OFFLINE. When a FILESTREAM file is offline, its parent filegroup will be internally marked as offline; therefore, all access to FILESTREAM data within that filegroup will fail.

Note

<add_or_modify_files> options are not available in a Contained Database.

<filespec>::=

Controls the file properties.

NAME logical_file_name

Specifies the logical name of the file.

logical_file_name

Is the logical name used in an instance of SQL Server when referencing the file.

NEWNAME new_logical_file_name

Specifies a new logical name for the file.

new_logical_file_name

Is the name to replace the existing logical file name. The name must be unique within the database and comply with the rules for identifiers. The name can be a character or Unicode constant, a regular identifier, or a delimited identifier.

FILENAME { 'os_file_name' | 'filestream_path' }

Specifies the operating system (physical) file name.

' os_file_name '

For a standard (ROWS) filegroup, this is the path and file name that is used by the operating system when you create the file. The file must reside on the server on which SQL Server is installed. The specified path must exist before executing the ALTER DATABASE statement.

SIZE, MAXSIZE, and FILEGROWTH parameters cannot be set when a UNC path is specified

for the file.

Note

System databases cannot reside on UNC share directories.

Data files should not be put on compressed file systems unless the files are read-only secondary files, or if the database is read-only. Log files should never be put on compressed file systems.

If the file is on a raw partition, os_file_name must specify only the drive letter of an existing raw partition. Only one file can be put on each raw partition.

'filestream_path'

For a FILESTREAM filegroup, FILENAME refers to a path where FILESTREAM data will be stored. The path up to the last folder must exist, and the last folder must not exist. For example, if you specify the path C:\MyFiles\MyFilestreamData, C:\MyFiles must exist before you run ALTER DATABASE, but the MyFilestreamData folder must not exist.

The filegroup and file (`<filespec>`) must be created in the same statement.

The SIZEand FILEGROWTH properties do not apply to a FILESTREAM filegroup.

SIZE size

Specifies the file size. SIZE does not apply to FILESTREAM filegroups.

size

Is the size of the file.

When specified with ADD FILE, size is the initial size for the file. When specified with MODIFY FILE, size is the new size for the file, and must be larger than the current file size.

When size is not supplied for the primary file, the SQL Server uses the size of the primary file in the **model** database. When a secondary data file or log file is specified but size is not specified for the file, the Database Engine makes the file 1 MB.

The KB, MB, GB, and TB suffixes can be used to specify kilobytes, megabytes, gigabytes, or terabytes. The default is MB. Specify a whole number and do not include a decimal. To specify a fraction of a megabyte, convert the value to kilobytes by multiplying the number by 1024. For example, specify 1536 KB instead of 1.5 MB (1.5 x 1024 = 1536).

MAXSIZE { max_size| UNLIMITED }

Specifies the maximum file size to which the file can grow.

max_size

Is the maximum file size. The KB, MB, GB, and TB suffixes can be used to specify kilobytes, megabytes, gigabytes, or terabytes. The default is MB. Specify a whole number and do not include a decimal. If max_size is not specified, the file size will increase until the disk is full.

UNLIMITED

Specifies that the file grows until the disk is full. In SQL Server, a log file specified with unlimited growth has a maximum size of 2 TB, and a data file has a maximum size of 16

TB. There is no maximum size when this option is specified for a FILESTREAM container. It continues to grow until the disk is full.

FILEGROWTH growth_increment

Specifies the automatic growth increment of the file. The FILEGROWTH setting for a file cannot exceed the MAXSIZE setting. FILEGROWTH does not apply to FILESTREAM filegroups.

growth_increment

Is the amount of space added to the file every time new space is required.

The value can be specified in MB, KB, GB, TB, or percent (%). If a number is specified without an MB, KB, or % suffix, the default is MB. When % is specified, the growth increment size is the specified percentage of the size of the file at the time the increment occurs. The size specified is rounded to the nearest 64 KB.

A value of 0 indicates that automatic growth is set to off and no additional space is allowed.

If FILEGROWTH is not specified, the default value is 1 MB for data files and 10% for log files, and the minimum value is 64 KB.

Note

Starting in SQL Server 2005, the default growth increment for data files has changed from 10% to 1 MB. The log file default of 10% remains unchanged.

OFFLINE

Sets the file offline and makes all objects in the filegroup inaccessible.

Caution

Use this option only when the file is corrupted and can be restored. A file set to OFFLINE can only be set online by restoring the file from backup. For more information about restoring a single file, see RESTORE (Transact-SQL).

Note

<filespec> options are not available in a Contained Database.

<add_or_modify_filegroups>::=

Add, modify, or remove a filegroup from the database.

ADD FILEGROUP filegroup_name

Adds a filegroup to the database.

CONTAINS FILESTREAM

Specifies that the filegroup stores FILESTREAM binary large objects (BLOBs) in the file system.

REMOVE FILEGROUP filegroup_name

Removes a filegroup from the database. The filegroup cannot be removed unless it is empty. Remove all files from the filegroup first. For more information, see "REMOVE FILE logical_file_name," earlier in this topic.

Note

Unless the FILESTREAM Garbage Collector has removed all the files from a FILESTREAM container, the ALTER DATABASE REMOVE FILE operation to remove a FILESTREAM container will fail and return an error. See the "Remove FILESTREAM Container" section in Remarks later in this topic.

MODIFY FILEGROUP filegroup_name { <filegroup_updatability_option> | DEFAULT | NAME = new_filegroup_name }

Modifies the filegroup by setting the status to READ_ONLY or READ_WRITE, making the filegroup the default filegroup for the database, or changing the filegroup name.

<filegroup_updatability_option>

Sets the read-only or read/write property to the filegroup.

DEFAULT

Changes the default database filegroup to filegroup_name. Only one filegroup in the database can be the default filegroup. For more information, see Understanding Files and Filegroups.

NAME = new_filegroup_name

Changes the filegroup name to the new_filegroup_name.

<filegroup_updatability_option>::=

Sets the read-only or read/write property to the filegroup.

READ_ONLY | READONLY

Specifies the filegroup is read-only. Updates to objects in it are not allowed. The primary filegroup cannot be made read-only. To change this state, you must have exclusive access to the database. For more information, see the SINGLE_USER clause.

Because a read-only database does not allow data modifications:

- Automatic recovery is skipped at system startup.
- Shrinking the database is not possible.
- No locking occurs in read-only databases. This can cause faster query performance.

Note

The keyword READONLY will be removed in a future version of Microsoft SQL Server. Avoid using READONLY in new development work, and plan to modify applications that currently use READONLY. Use READ_ONLY instead.

READ_WRITE | READWRITE

Specifies the group is READ_WRITE. Updates are enabled for the objects in the filegroup. To change this state, you must have exclusive access to the database. For more information, see the SINGLE_USER clause.

Note

The keyword READWRITE will be removed in a future version of Microsoft SQL Server. Avoid using

READWRITE in new development work, and plan to modify applications that currently use READWRITE. Use READ_WRITE instead.

The status of these options can be determined by examining the **is_read_only** column in the **sys.databases** catalog view or the **Updateability** property of the DATABASEPROPERTYEX function.

Remarks

To decrease the size of a database, use DBCC SHRINKDATABASE.

You cannot add or remove a file while a BACKUP statement is running.

A maximum of 32,767 files and 32,767 filegroups can be specified for each database.

In SQL Server 2005 or later, the state of a database file (for example, online or offline), is maintained independently from the state of the database. For more information, see File States. The state of the files within a filegroup determines the availability of the whole filegroup. For a filegroup to be available, all files within the filegroup must be online. If a filegroup is offline, any try to access the filegroup by an SQL statement will fail with an error. When you build query plans for SELECT statements, the query optimizer avoids nonclustered indexes and indexed views that reside in offline filegroups. This enables these statements to succeed. However, if the offline filegroup contains the heap or clustered index of the target table, the SELECT statements fail. Additionally, any INSERT, UPDATE, or DELETE statement that modifies a table with any index in an offline filegroup will fail.

Moving Files

In SQL Server 2005 or later, you can move system or user-defined data and log files by specifying the new location in FILENAME. This may be useful in the following scenarios:

- Failure recovery. For example, the database is in suspect mode or shutdown caused by hardware failure
- Planned relocation
- Relocation for scheduled disk maintenance

For more information, see Moving Database Files.

Initializing Files

By default, data and log files are initialized by filling the files with zeros when you perform one of the following operations:

- Create a database
- Add files to an existing database
- Increase the size of an existing file
- Restore a database or filegroup

Data files can be initialized instantaneously. This enables for fast execution of these file operations.

Removing a FILESTREAM Container

Even though FILESTREAM container may have been emptied using the "DBCC SHRINKFILE" operation, the database may still need to maintain references to the deleted files for various system maintenance reasons. sp_filestream_force_garbage_collection (Transact-SQL) will run the FILESTREAM Garbage Collector to remove these files when it is safe to do so. Unless the FILESTREAM Garbage Collector has removed all the files from a FILESTREAM container, the ALTER DATABASEREMOVE FILE operation will fail to remove a FILESTREAM container and will return an error. The following process is recommended to remove a FILESTREAM container.

1. Run DBCC SHRINKFILE with the EMPTYFILE option to move the active contents of this container to other containers.
2. Ensure that Log backups have been taken, in the FULL or BULK_LOGGED recovery model.
3. Ensure that the replication log reader job has been run, if relevant.
4. Run sp_filestream_force_garbage_collection to force the garbage collector to delete any files that are no longer needed in this container.
5. Execute ALTER DATABASE with the REMOVE FILE option to remove this container.
6. Repeat steps 2 through 4 once more to complete the garbage collection.
7. Use ALTER Database...REMOVE FILE to remove this container.

Examples

A. Adding a file to a database

The following example adds a 5-MB data file to the `AdventureWorks2012` database.

```
USE master;
GO
ALTER DATABASE AdventureWorks2012
ADD FILE
(
    NAME = Test1dat2,
    FILENAME = 'C:\Program Files\Microsoft SQL
Server\MSSQL10_50.MSSQLSERVER\MSSQL\DATA\t1dat2.ndf',
    SIZE = 5MB,
    MAXSIZE = 100MB,
    FILEGROWTH = 5MB
);
GO
```

B. Adding a filegroup with two files to a database

The following example creates the filegroup `Test1FG1` in the `AdventureWorks2012` database and adds two 5-MB files to the filegroup.

```
USE master
GO
```

```
ALTER DATABASE AdventureWorks2012
ADD FILEGROUP Test1FG1;
GO
ALTER DATABASE AdventureWorks2012
ADD FILE
(
    NAME = test1dat3,
    FILENAME = 'C:\Program Files\Microsoft SQL
Server\MSSQL10_50.MSSQLSERVER\MSSQL\DATA\t1dat3.ndf',
    SIZE = 5MB,
    MAXSIZE = 100MB,
    FILEGROWTH = 5MB
),
(
    NAME = test1dat4,
    FILENAME = 'C:\Program Files\Microsoft SQL
Server\MSSQL10_50.MSSQLSERVER\MSSQL\DATA\t1dat4.ndf',
    SIZE = 5MB,
    MAXSIZE = 100MB,
    FILEGROWTH = 5MB
)
TO FILEGROUP Test1FG1;
GO
```

C. Adding two log files to a database

The following example adds two 5-MB log files to the `AdventureWorks2012` database.

```
USE master;
GO
ALTER DATABASE AdventureWorks2012
ADD LOG FILE
(
    NAME = test1log2,
    FILENAME = 'C:\Program Files\Microsoft SQL
Server\MSSQL10_50.MSSQLSERVER\MSSQL\DATA\test2log.ldf',
    SIZE = 5MB,
    MAXSIZE = 100MB,
```

```
    FILEGROWTH = 5MB
),
(
    NAME = test1log3,
    FILENAME = 'C:\Program Files\Microsoft SQL
Server\MSSQL10_50.MSSQLSERVER\MSSQL\DATA\test3log.ldf',
    SIZE = 5MB,
    MAXSIZE = 100MB,
    FILEGROWTH = 5MB
);
GO
```

D. Removing a file from a database

The following example removes one of the files added in example B.

```
USE master;
GO
ALTER DATABASE AdventureWorks2012
REMOVE FILE test1dat4;
GO
```

E. Modifying a file

The following example increases the size of one of the files added in example B.

```
USE master;
GO
ALTER DATABASE AdventureWorks2012
MODIFY FILE
    (NAME = test1dat3,
    SIZE = 20MB);
GO
```

F. Moving a file to a new location

The following example moves the `Test1dat2` file created in example A to a new directory.

Note

You must physically move the file to the new directory before running this example. Afterward, stop and start the instance of SQL Server or take the database OFFLINE and then ONLINE to implement the change.

```
USE master;
GO
ALTER DATABASE AdventureWorks2012
MODIFY FILE
(
    NAME = Test1dat2,
    FILENAME = N'c:\t1dat2.ndf'
);
GO
```

G. Moving tempdb to a new location

The following example moves `tempdb` from its current location on the disk to another disk location. Because `tempdb` is re-created each time the MSSQLSERVER service is started, you do not have to physically move the data and log files. The files are created when the service is restarted in step 3. Until the service is restarted, `tempdb` continues to function in its existing location.

1. Determine the logical file names of the `tempdb` database and their current location on disk.

```
SELECT name, physical_name
FROM sys.master_files
WHERE database_id = DB_ID('tempdb');
GO
```

2. Change the location of each file by using `ALTER DATABASE`.

```
USE master;
GO
ALTER DATABASE tempdb
MODIFY FILE (NAME = tempdev, FILENAME = 'E:\SQLData\tempdb.mdf');
GO
ALTER DATABASE  tempdb
MODIFY FILE (NAME = templog, FILENAME = 'E:\SQLData\templog.ldf');
GO
```

3. Stop and restart the instance of SQL Server.
4. Verify the file change.

```
SELECT name, physical_name
```

```
    FROM sys.master_files
    WHERE database_id = DB_ID('tempdb');
```

5. Delete the tempdb.mdf and templog.ldf files from their original location.

H. Making a filegroup the default

The following example makes the `Test1FG1` filegroup created in example B the default filegroup. Then, the default filegroup is reset to the `PRIMARY` filegroup. Note that `PRIMARY` must be delimited by brackets or quotation marks.

```
USE master;
GO
ALTER DATABASE AdventureWorks2012
MODIFY FILEGROUP Test1FG1 DEFAULT;
GO
ALTER DATABASE AdventureWorks2012
MODIFY FILEGROUP [PRIMARY] DEFAULT;
GO
```

I. Adding a Filegroup Using ALTER DATABASE

The following example adds a `FILEGROUP` that contains the `FILESTREAM` clause to the `FileStreamPhotoDB` database.

```
--Create and add a FILEGROUP that CONTAINS the FILESTREAM clause to
--the FileStreamPhotoDB database.
ALTER database FileStreamPhotoDB
ADD FILEGROUP TodaysPhotoShoot
CONTAINS FILESTREAM
GO

--Add a file for storing database photos to FILEGROUP
ALTER database FileStreamPhotoDB
ADD FILE
(
    NAME= 'PhotoShoot1',
    FILENAME = 'C:\Users\Administrator\Pictures\TodaysPhotoShoot.ndf'
)
TO FILEGROUP TodaysPhotoShoot
GO
```

See Also

CREATE DATABASE

DATABASEPROPERTYEX

DROP DATABASE

sp_spaceused

sys.databases (Transact-SQL)

sys.database_files

sys.data_spaces (Transact-SQL)

sys.filegroups

sys.master_files (Transact-SQL)

Designing and Implementing FILESTREAM Storage

DBCC SHRINKFILE

sp_filestream_force_garbage_collection

ALTER DATABASE SET Options

This topic contains the ALTER DATABASE syntax that is related to setting database options. For other ALTER DATABASE syntax, see ALTER DATABASE (Transact-SQL). Database mirroring, AlwaysOn Availability Groups, and compatibility levels are SET options but are described in separate topics because of their length. For more information, see ALTER DATABASE Database Mirroring (Transact-SQL), ALTER DATABASE SET HADR (Transact-SQL), and ALTER DATABASE Compatibility Level (Transact-SQL).

 Transact-SQL Syntax Conventions

Syntax

```
ALTER DATABASE { database_name  | CURRENT }
SET
{
   <optionspec> [ ,... n ] [ WITH <termination> ]
}

<optionspec> ::=
{
   <auto_option>
  | <change_tracking_option>
  | <containment_option>
```

```
  | <cursor_option>
  | <database_mirroring_option>
  | <date_correlation_optimization_option>
  | <db_encryption_option>
  | <db_state_option>
  | <db_update_option>
  | <db_user_access_option>
  | <external_access_option>
  | FILESTREAM ( <FILESTREAM_option> )
  | <HADR_options>
  | <parameterization_option>
  | <recovery_option>
  | <target_recovery_time_option>
  | <service_broker_option>
  | <snapshot_option>
  | <sql_option>
}

<auto_option> ::=
{
    AUTO_CLOSE { ON | OFF }
  | AUTO_CREATE_STATISTICS { ON | OFF }
  | AUTO_SHRINK { ON | OFF }
  | AUTO_UPDATE_STATISTICS { ON | OFF }
  | AUTO_UPDATE_STATISTICS_ASYNC { ON | OFF }
}

<change_tracking_option> ::=
{
  CHANGE_TRACKING
   {
      = OFF
    | = ON [ ( <change_tracking_option_list > [,... n] ) ]
    | ( <change_tracking_option_list> [,... n ] )
   }
```

}

<change_tracking_option_list> ::=
{
AUTO_CLEANUP = { ON | OFF }
| CHANGE_RETENTION = retention_period { DAYS | HOURS | MINUTES }
}

<containment_option> ::=
CONTAINMENT = { NONE | PARTIAL }

<cursor_option> ::=
{
CURSOR_CLOSE_ON_COMMIT { ON | OFF }
| CURSOR_DEFAULT { LOCAL | GLOBAL }
}

<database_mirroring_option>
ALTER DATABASE Database Mirroring

<date_correlation_optimization_option> ::=
DATE_CORRELATION_OPTIMIZATION { ON | OFF }

<db_encryption_option> ::=
ENCRYPTION { ON | OFF }

<db_state_option> ::=
{ ONLINE | OFFLINE | EMERGENCY }

<db_update_option> ::=
{ READ_ONLY | READ_WRITE }

<db_user_access_option> ::=
{ SINGLE_USER | RESTRICTED_USER | MULTI_USER }

<external_access_option> ::=

```
{
    DB_CHAINING { ON | OFF }
  | TRUSTWORTHY { ON | OFF }
  | DEFAULT_FULLTEXT_LANGUAGE = { <lcid> | <language name> | <language alias> }
  | DEFAULT_LANGUAGE = { <lcid> | <language name> | <language alias> }
  | NESTED_TRIGGERS = { OFF | ON }
  | TRANSFORM_NOISE_WORDS = { OFF | ON }
  | TWO_DIGIT_YEAR_CUTOFF = { 1753, ..., 2049, ..., 9999 }
}
```

<FILESTREAM_option> ::=

```
{
    NON_TRANSACTED_ACCESS = { OFF | READ_ONLY | FULL }
  | DIRECTORY_NAME = <directory_name>
}
```

<HADR_options> ::=

```
 ALTER DATABASE SET HADR
```

<parameterization_option> ::=

```
    PARAMETERIZATION { SIMPLE | FORCED }
```

<recovery_option> ::=

```
{
    RECOVERY { FULL | BULK_LOGGED | SIMPLE }
  | TORN_PAGE_DETECTION { ON | OFF }
  | PAGE_VERIFY { CHECKSUM | TORN_PAGE_DETECTION | NONE }
}
```

<target_recovery_time_option> ::=

```
    TARGET_RECOVERY_TIME = target_recovery_time { SECONDS | MINUTES }
```

<service_broker_option> ::=

```
{
    ENABLE_BROKER
  | DISABLE_BROKER
```

```
  | NEW_BROKER
  | ERROR_BROKER_CONVERSATIONS
  | HONOR_BROKER_PRIORITY { ON | OFF}
}
```

<snapshot_option> ::=

```
{
    ALLOW_SNAPSHOT_ISOLATION { ON | OFF }
  | READ_COMMITTED_SNAPSHOT {ON | OFF }
}
```

<sql_option> ::=

```
{
    ANSI_NULL_DEFAULT { ON | OFF }
  | ANSI_NULLS { ON | OFF }
  | ANSI_PADDING { ON | OFF }
  | ANSI_WARNINGS { ON | OFF }
  | ARITHABORT { ON | OFF }
  | COMPATIBILITY_LEVEL = { 90 | 100 | 110 }
  | CONCAT_NULL_YIELDS_NULL { ON | OFF }
  | NUMERIC_ROUNDABORT { ON | OFF }
  | QUOTED_IDENTIFIER { ON | OFF }
  | RECURSIVE_TRIGGERS { ON | OFF }
}
```

<termination> ::=

```
{
    ROLLBACK AFTER integer [ SECONDS ]
  | ROLLBACK IMMEDIATE
  | NO_WAIT
}
```

Arguments

database_name | CURRENT

Is the name of the database to be modified. CURRENT performs the action in the current database. CURRENT is not supported for all options in all contexts. If CURRENT fails, provide the database name.

<auto_option> ::=

Controls automatic options.

AUTO_CLOSE { ON | OFF }

ON

The database is shut down cleanly and its resources are freed after the last user exits.

The database automatically reopens when a user tries to use the database again. For example, by issuing a USE database_name statement. If the database is shut down cleanly while AUTO_CLOSE is set to ON, the database is not reopened until a user tries to use the database the next time the Database Engine is restarted.

OFF

The database remains open after the last user exits.

The AUTO_CLOSE option is useful for desktop databases because it allows for database files to be managed as regular files. They can be moved, copied to make backups, or even e-mailed to other users. The AUTO_CLOSE process is asynchronous; repeatedly opening and closing the database does not reduce performance.

Note

The AUTO_CLOSE option is not available in a Contained Database.

The status of this option can be determined by examining the is_auto_close_on column in the sys.databases catalog view or the IsAutoClose property of the DATABASEPROPERTYEX function.

Note

When AUTO_CLOSE is ON, some columns in the sys.databases catalog view and DATABASEPROPERTYEX function will return NULL because the database is unavailable to retrieve the data. To resolve this, execute a USE statement to open the database.

Note

Database mirroring requires AUTO_CLOSE OFF.

When the database is set to AUTOCLOSE = ON, an operation that initiates an automatic database shutdown clears the plan cache for the instance of SQL Server. Clearing the plan cache causes a recompilation of all subsequent execution plans and can cause a sudden, temporary decrease in query performance. In SQL Server 2005 Service Pack 2 and higher, for each cleared cachestore in the plan cache, the SQL Server error log contains the following informational message: "SQL Server has encountered %d occurrence(s) of cachestore flush for the '%s' cachestore (part of plan cache) due to some database maintenance or reconfigure operations". This message is logged every five minutes as long as the cache is flushed within that time interval.

AUTO_CREATE_STATISTICS { ON | OFF }

ON

The query optimizer creates statistics on single columns in query predicates, as necessary, to improve query plans and query performance. These single-column statistics are created when the query optimizer compiles queries. The single-column statistics are created only

on columns that are not already the first column of an existing statistics object.

The default is ON. We recommend that you use the default setting for most databases.

OFF

The query optimizer does not create statistics on single columns in query predicates when it is compiling queries. Setting this option to OFF can cause suboptimal query plans and degraded query performance.

The status of this option can be determined by examining the is_auto_create_stats_on column in the sys.databases catalog view or the IsAutoCreateStatistics property of the DATABASEPROPERTYEX function.

For more information, see the section "Using the Database-Wide Statistics Options" in Using Statistics to Improve Query Performance.

AUTO_SHRINK { ON | OFF }

ON

The database files are candidates for periodic shrinking.

Both data file and log files can be automatically shrunk. AUTO_SHRINK reduces the size of the transaction log only if the database is set to SIMPLE recovery model or if the log is backed up. When set to OFF, the database files are not automatically shrunk during periodic checks for unused space.

The AUTO_SHRINK option causes files to be shrunk when more than 25 percent of the file contains unused space. The file is shrunk to a size where 25 percent of the file is unused space, or to the size of the file when it was created, whichever is larger.

You cannot shrink a read-only database.

OFF

The database files are not automatically shrunk during periodic checks for unused space.

The status of this option can be determined by examining the is_auto_shrink_on column in the sys.databases catalog view or the IsAutoShrink property of the DATABASEPROPERTYEX function.

Note

The AUTO_SHRINK option is not available in a Contained Database.

AUTO_UPDATE_STATISTICS { ON | OFF }

ON

Specifies that the query optimizer updates statistics when they are used by a query and when they might be out-of-date. Statistics become out-of-date after insert, update, delete, or merge operations change the data distribution in the table or indexed view. The query optimizer determines when statistics might be out-of-date by counting the number of data modifications since the last statistics update and comparing the number of modifications to a threshold. The threshold is based on the number of rows in the table or indexed view.

The query optimizer checks for out-of-date statistics before compiling a query and before executing a cached query plan. Before compiling a query, the query optimizer uses the columns, tables, and indexed views in the query predicate to determine which statistics might be out-of-date. Before executing a cached query plan, the Database Engine verifies that the query plan references up-to-date statistics.

The AUTO_UPDATE_STATISTICS option applies to statistics created for indexes, single-columns in query predicates, and statistics that are created by using the CREATE STATISTICS statement. This option also applies to filtered statistics.

The default is ON. We recommend that you use the default setting for most databases.

Use the AUTO_UPDATE_STATISTICS_ASYNC option to specify whether the statistics are updated synchronously or asynchronously.

OFF

Specifies that the query optimizer does not update statistics when they are used by a query and when they might be out-of-date. Setting this option to OFF can cause suboptimal query plans and degraded query performance.

The status of this option can be determined by examining the is_auto_update_stats_on column in the sys.databases catalog view or the IsAutoUpdateStatistics property of the DATABASEPROPERTYEX function.

For more information, see the section "Using the Database-Wide Statistics Options" in Using Statistics to Improve Query Performance.

AUTO_UPDATE_STATISTICS_ASYNC { ON | OFF }

ON

Specifies that statistics updates for the AUTO_UPDATE_STATISTICS option are asynchronous. The query optimizer does not wait for statistics updates to complete before it compiles queries.

Setting this option to ON has no effect unless AUTO_UPDATE_STATISTICS is set to ON.

By default, the AUTO_UPDATE_STATISTICS_ASYNC option is set to OFF, and the query optimizer updates statistics synchronously.

OFF

Specifies that statistics updates for the AUTO_UPDATE_STATISTICS option are synchronous. The query optimizer waits for statistcs updates to complete before it compiles queries.

Setting this option to OFF has no effect unless AUTO_UPDATE_STATISTICS is set to ON.

The status of this option can be determined by examining the is_auto_update_stats_async_on column in the sys.databases catalog view.

For more information that describes when to use synchronous or asynchronous statistics updates, see the section "Using the Database-Wide Statistics Options" in Using Statistics to Improve Query Performance.

<change_tracking_option> ::=

Controls change tracking options. You can enable change tracking, set options, change options, and disable change tracking. For examples, see the Examples section later in this topic.

ON

Enables change tracking for the database. When you enable change tracking, you can also set the AUTO CLEANUP and CHANGE RETENTION options.

AUTO_CLEANUP = { ON | OFF }

ON

Change tracking information is automatically removed after the specified retention period.

OFF

Change tracking data is not removed from the database.

CHANGE_RETENTION = retention_period { DAYS | HOURS | MINUTES }

Specifies the minimum period for keeping change tracking information in the database. Data is removed only when the AUTO_CLEANUP value is ON.

retention_period is an integer that specifies the numerical component of the retention period.

The default retention period is 2 days. The minimum retention period is 1 minute.

OFF

Disables change tracking for the database. You must disable change tracking on all tables before you can disable change tracking off the database.

<containment_option> ::=

Controls database containment options.

CONTAINMENT = { NONE | PARTIAL}

NONE

The database is not a contained database.

PARTIAL

The database is a contained database. Setting database containment to partial will fail if the database has replication, change data capture, or change tracking enabled. Error checking stops after one failure. For more information about contained databases, see Understanding Contained Databases.

<cursor_option> ::=

Controls cursor options.

CURSOR_CLOSE_ON_COMMIT { ON | OFF }

ON

Any cursors open when a transaction is committed or rolled back are closed.

OFF

Cursors remain open when a transaction is committed; rolling back a transaction closes any cursors except those defined as INSENSITIVE or STATIC.

Connection-level settings that are set by using the SET statement override the default database setting for CURSOR_CLOSE_ON_COMMIT. By default, ODBC and OLE DB clients issue a connection-level SET statement setting CURSOR_CLOSE_ON_COMMIT to OFF for the session when connecting to an instance of SQL Server. For more information, see SET CURSOR_CLOSE_ON_COMMIT (Transact-SQL).

The status of this option can be determined by examining the is_cursor_close_on_commit_on column in the sys.databases catalog view or the IsCloseCursorsOnCommitEnabled property of the DATABASEPROPERTYEX function.

CURSOR_DEFAULT { LOCAL | GLOBAL }

Controls whether cursor scope uses LOCAL or GLOBAL.

LOCAL

When LOCAL is specified and a cursor is not defined as GLOBAL when created, the scope of the cursor is local to the batch, stored procedure, or trigger in which the cursor was created. The cursor name is valid only within this scope. The cursor can be referenced by local cursor variables in the batch, stored procedure, or trigger, or a stored procedure OUTPUT parameter. The cursor is implicitly deallocated when the batch, stored procedure, or trigger ends, unless it was passed back in an OUTPUT parameter. If the cursor is passed back in an OUTPUT parameter, the cursor is deallocated when the last variable that references it is deallocated or goes out of scope.

GLOBAL

When GLOBAL is specified, and a cursor is not defined as LOCAL when created, the scope of the cursor is global to the connection. The cursor name can be referenced in any stored procedure or batch executed by the connection.

The cursor is implicitly deallocated only at disconnect. For more information, see DECLARE CURSOR.

The status of this option can be determined by examining the is_local_cursor_default column in the sys.databases catalog view or the IsLocalCursorsDefault property of the DATABASEPROPERTYEX function.

<database_mirroring>

For the argument descriptions, see ALTER DATABASE Database Mirroring (Transact-SQL).

<date_correlation_optimization_option> ::=

Controls the date_correlation_optimization option.

DATE_CORRELATION_OPTIMIZATION { ON | OFF }

ON

SQL Server maintains correlation statistics between any two tables in the database that are linked by a FOREIGN KEY constraint and have **datetime** columns.

OFF

Correlation statistics are not maintained.

To set DATE_CORRELATION_OPTIMIZATION to ON, there must be no active connections to the database except for the connection that is executing the ALTER DATABASE statement. Afterwards, multiple connections are supported.

The current setting of this option can be determined by examining the is_date_correlation_on column in the sys.databases catalog view.

<db_encryption_option> ::=

Controls the database encryption state.

ENCRYPTION {ON | OFF}

Sets the database to be encrypted (ON) or not encrypted (OFF). For more information about database encryption, see Understanding Transparent Data Encryption (TDE).

When encryption is enabled at the database level all filegroups will be encrypted. Any new filegroups will inherit the encrypted property. If any filegroups in the database are set to **READ ONLY**, the database encryption operation will fail.

You can see the encryption state of the database by using the sys.dm_database_encryption_keys dynamic management view.

<db_state_option> ::=

Controls the state of the database.

OFFLINE

The database is closed, shut down cleanly, and marked offline. The database cannot be modified while it is offline.

ONLINE

The database is open and available for use.

EMERGENCY

The database is marked READ_ONLY, logging is disabled, and access is limited to members of the sysadmin fixed server role. EMERGENCY is primarily used for troubleshooting purposes. For example, a database marked as suspect due to a corrupted log file can be set to the EMERGENCY state. This could enable the system administrator read-only access to the database. Only members of the sysadmin fixed server role can set a database to the EMERGENCY state.

Note

Permissions: ALTER DATABASE permission for the subject database is required to change a database to the offline or emergency state. The server level ALTER ANY DATABASE permission is required to move a database from offline to online.

The status of this option can be determined by examining the state and state_desc columns in the sys.databases catalog view or the Status property of the DATABASEPROPERTYEX function. For more information, see Database States.

A database marked as RESTORING cannot be set to OFFLINE, ONLINE, or EMERGENCY. A database may be in the RESTORING state during an active restore operation or when a restore operation of a database or log file fails because of a corrupted backup file.

<db_update_option> ::=

Controls whether updates are allowed on the database.

READ_ONLY

Users can read data from the database but not modify it.

Note

To improve query performance, update statistics before setting a database to READ_ONLY. If additional statistics are needed after a database is set to READ_ONLY, the Database Engine will create statistics in tempdb. For more information about statistics for a read-only database, see Statistics.

READ_WRITE

The database is available for read and write operations.

To change this state, you must have exclusive access to the database. For more information, see the SINGLE_USER clause.

<db_user_access_option> ::=

Controls user access to the database.

SINGLE_USER

Specifies that only one user at a time can access the database. If SINGLE_USER is specified and there are other users connected to the database the ALTER DATABASE statement will be blocked until all users disconnect from the specified database. To override this behavior, see the WITH <termination> clause.

The database remains in SINGLE_USER mode even if the user that set the option logs off. At that point, a different user, but only one, can connect to the database.

Before you set the database to SINGLE_USER, verify the AUTO_UPDATE_STATISTICS_ASYNC option is set to OFF. When set to ON, the background thread used to update statistics takes a connection against the database, and you will be unable to access the database in single-user mode. To view the status of this option, query the is_auto_update_stats_async_on column in the sys.databases catalog view. If the option is set to ON, perform the following tasks:

1. Set AUTO_UPDATE_STATISTICS_ASYNC to OFF.
2. Check for active asynchronous statistics jobs by querying the sys.dm_exec_background_job_queue dynamic management view.

If there are active jobs, either allow the jobs to complete or manually terminate them by using KILL STATS JOB.

RESTRICTED_USER

RESTRICTED_USER allows for only members of the db_owner fixed database role and dbcreator and sysadmin fixed server roles to connect to the database, but does not limit their number. All connections to the database are disconnected in the timeframe specified by the termination clause of the ALTER DATABASE statement. After the database has transitioned to the RESTRICTED_USER state, connection attempts by unqualified users are refused.

MULTI_USER

All users that have the appropriate permissions to connect to the database are allowed.

The status of this option can be determined by examining the user_access column in the sys.databases catalog view or the UserAccess property of the DATABASEPROPERTYEX function.

<external_access_option> ::=

Controls whether the database can be accessed by external resources, such as objects from another database.

DB_CHAINING { ON | OFF }

ON

Database can be the source or target of a cross-database ownership chain.

OFF

Database cannot participate in cross-database ownership chaining.

Important

The instance of SQL Server will recognize this setting when the cross db ownership chaining server option is 0 (OFF). When cross db ownership chaining is 1 (ON), all user databases can participate in cross-database ownership chains, regardless of the value of this option. This option is set by using sp_configure.

To set this option, requires CONTROL SERVER permission on the database.

The DB_CHAINING option cannot be set on these system databases: master, model, and tempdb.

The status of this option can be determined by examining the is_db_chaining_on column in the sys.databases catalog view.

TRUSTWORTHY { ON | OFF }

ON

Database modules (for example, user-defined functions or stored procedures) that use an impersonation context can access resources outside the database.

OFF

Database modules in an impersonation context cannot access resources outside the database.

TRUSTWORTHY is set to OFF whenever the database is attached.

By default, all system databases except the msdb database have TRUSTWORTHY set to OFF. The value cannot be changed for the model and tempdb databases. We recommend that you never set the TRUSTWORTHY option to ON for the master database.

To set this option, requires CONTROL SERVER permission on the database.

The status of this option can be determined by examining the is_trustworthy_on column in the sys.databases catalog view.

DEFAULT_FULLTEXT_LANGUAGE

Specifies the default language value for full-text indexed columns.

Important

This option is allowable only when CONTAINMENT has been set to PARTIAL. If CONTAINMENT is set to NONE, errors will occur.

DEFAULT_LANGUAGE

Specifies the default language for all newly created logins. Language can be specified by

providing the local id (lcid), the language name, or the language alias. For a list of acceptable language names and aliases, see sys.syslanguages (Transact-SQL).

Important
This option is allowable only when CONTAINMENT has been set to PARTIAL. If CONTAINMENT is set to NONE, errors will occur.

NESTED_TRIGGERS

Specifies whether an AFTER trigger can cascade; that is, perform an action that initiates another trigger, which initiates another trigger, and so on.

Important
This option is allowable only when CONTAINMENT has been set to PARTIAL. If CONTAINMENT is set to NONE, errors will occur.

TRANSFORM_NOISE_WORDS

Used to suppress an error message if noise words, or stopwords, cause a Boolean operation on a full-text query to fail.

Important
This option is allowable only when CONTAINMENT has been set to PARTIAL. If CONTAINMENT is set to NONE, errors will occur.

TWO_DIGIT_YEAR_CUTOFF

Specifies an integer from 1753 to 9999 that represents the cutoff year for interpreting two-digit years as four-digit years.

Important
This option is allowable only when CONTAINMENT has been set to PARTIAL. If CONTAINMENT is set to NONE, errors will occur.

<FILESTREAM_option> ::=

Controls the settings for FileTables.

NON_TRANSACTED_ACCESS = { OFF | READ_ONLY | FULL }

OFF

Non-transactional access to FileTable data is disabled.

READ_ONLY

FILESTREAM data in FileTables in this database can be read by non-transactional processes.

FULL

Full non-transactional access to FILESTREAM data in FileTables is enabled.

DIRECTORY_NAME = <directory_name>

A windows-compatible directory name. This name should be unique among all the database-level directory names in the SQL Server instance. Uniqueness comparison is case-insensitive,

regardless of collation settings. This option must be set before creating a FileTable in this database.

<parameterization_option> ::=

Controls the parameterization option.

PARAMETERIZATION { SIMPLE | FORCED }

SIMPLE

Queries are parameterized based on the default behavior of the database.

FORCED

SQL Server parameterizes all queries in the database.

The current setting of this option can be determined by examining the is_parameterization_forced column in the sys.databases catalog view.

<recovery_option> ::=

Controls database recovery options and disk I/O error checking.

FULL

Provides full recovery after media failure by using transaction log backups. If a data file is damaged, media recovery can restore all committed transactions. For more information, see Recovery Models (SQL Server).

BULK_LOGGED

Provides recovery after media failure by combining the best performance and least amount of log-space use for certain large-scale or bulk operations. For information about what operations can be minimally logged, see Transaction Logs (SQL Server). Under the BULK_LOGGED recovery model, logging for these operations is minimal. For more information, see Recovery Models (SQL Server).

SIMPLE

A simple backup strategy that uses minimal log space is provided. Log space can be automatically reused when it is no longer required for server failure recovery. For more information, see Recovery Models (SQL Server).

Important

The simple recovery model is easier to manage than the other two models but at the expense of greater data loss exposure if a data file is damaged. All changes since the most recent database or differential database backup are lost and must be manually reentered.

The default recovery model is determined by the recovery model of the **model** database. For more information about selecting the appropriate recovery model, see Database Recovery Models (SQL Server).

The status of this option can be determined by examining the **recovery_model** and **recovery_model_desc** columns in the sys.databases catalog view or the Recovery property of the DATABASEPROPERTYEX function.

TORN_PAGE_DETECTION { ON | OFF }

ON

Incomplete pages can be detected by the Database Engine.

OFF

Incomplete pages cannot be detected by the Database Engine.

Important

The syntax structure TORN_PAGE_DETECTION ON | OFF will be removed in a future version of SQL Server. Avoid using this syntax structure in new development work, and plan to modify applications that currently use the syntax structure. Use the PAGE_VERIFY option instead.

PAGE_VERIFY { CHECKSUM | TORN_PAGE_DETECTION | NONE }

Discovers damaged database pages caused by disk I/O path errors. Disk I/O path errors can be the cause of database corruption problems and are generally caused by power failures or disk hardware failures that occur at the time the page is being written to disk.

CHECKSUM

Calculates a checksum over the contents of the whole page and stores the value in the page header when a page is written to disk. When the page is read from disk, the checksum is recomputed and compared to the checksum value stored in the page header. If the values do not match, error message 824 (indicating a checksum failure) is reported to both the SQL Server error log and the Windows event log. A checksum failure indicates an I/O path problem. To determine the root cause requires investigation of the hardware, firmware drivers, BIOS, filter drivers (such as virus software), and other I/O path components.

TORN_PAGE_DETECTION

Saves a specific 2-bit pattern for each 512-byte sector in the 8-kilobyte (KB) database page and stored in the database page header when the page is written to disk. When the page is read from disk, the torn bits stored in the page header are compared to the actual page sector information. Unmatched values indicate that only part of the page was written to disk. In this situation, error message 824 (indicating a torn page error) is reported to both the SQL Server error log and the Windows event log. Torn pages are typically detected by database recovery if it is truly an incomplete write of a page. However, other I/O path failures can cause a torn page at any time.

NONE

Database page writes will not generate a CHECKSUM or TORN_PAGE_DETECTION value. SQL Server will not verify a checksum or torn page during a read even if a CHECKSUM or TORN_PAGE_DETECTION value is present in the page header.

Consider the following important points when you use the PAGE_VERIFY option:

- The default is CHECKSUM.
- When a user or system database is upgraded to SQL Server 2005 or a later version, the

PAGE_VERIFY value (NONE or TORN_PAGE_DETECTION) is retained. We recommend that you use CHECKSUM.

Note

In earlier versions of SQL Server, the PAGE_VERIFY database option is set to NONE for the tempdb database and cannot be modified. In SQL Server 2008 and later versions, the default value for the tempdb database is CHECKSUM for new installations of SQL Server. When upgrading an installation SQL Server, the default value remains NONE. The option can be modified. We recommend that you use CHECKSUM for the tempdb database.

- TORN_PAGE_DETECTION may use fewer resources but provides a minimal subset of the CHECKSUM protection.
- PAGE_VERIFY can be set without taking the database offline, locking the database, or otherwise impeding concurrency on that database.
- CHECKSUM is mutually exclusive to TORN_PAGE_DETECTION. Both options cannot be enabled at the same time.

When a torn page or checksum failure is detected, you can recover by restoring the data or potentially rebuilding the index if the failure is limited only to index pages. If you encounter a checksum failure, to determine the type of database page or pages affected, run DBCC CHECKDB. For more information about restore options, see RESTORE Arguments (Transact-SQL). Although restoring the data will resolve the data corruption problem, the root cause, for example, disk hardware failure, should be diagnosed and corrected as soon as possible to prevent continuing errors.

SQL Server will retry any read that fails with a checksum, torn page, or other I/O error four times. If the read is successful in any one of the retry attempts, a message will be written to the error log and the command that triggered the read will continue. If the retry attempts fail, the command will fail with error message 824.

For more information about checksum, torn page, read-retry, error messages 823 and 824, and other SQL Server I/O auditing features, see this Microsoft Web site.

The current setting of this option can be determined by examining the page_verify_option column in the sys.databases catalog view or the IsTornPageDetectionEnabled property of the DATABASEPROPERTYEX function.

<target_recovery_time_option> ::=

Specifies the frequency of indirect checkpoints on a per-database basis. The default is 0, which indicates that the database will use automatic checkpoints, whose frequency depends on the recovery interval setting of the server instance.

TARGET_RECOVERY_TIME = target_recovery_time { SECONDS | MINUTES }

target_recovery_time

Specifies the maximum bound on the time to recover the specified database in the event of a crash.

SECONDS

Indicates that target_recovery_time is expressed as the number of seconds.

MINUTES

Indicates that target_recovery_time is expressed as the number of minutes.

For more information about indirect checkpoints, see Database Checkpoints (SQL Server).

<service_broker_option> ::=

Controls the following Service Broker options: enables or disables message delivery, sets a new Service Broker identifier, or sets conversation priorities to ON or OFF.

ENABLE_BROKER

Specifies that Service Broker is enabled for the specified database. Message delivery is started, and the is_broker_enabled flag is set to true in the sys.databases catalog view. The database retains the existing Service Broker identifier.

Note

ENABLE_BROKER requires an exclusive database lock. If other sessions have locked resources in the database, ENABLE_BROKER will wait until the other sessions release their locks. To enable Service Broker in a user database, ensure that no other sessions are using the database before you run the ALTER DATABASE SET ENABLE_BROKER statement, such as by putting the database in single user mode. To enable Service Broker in the msdb database, first stop SQL Server Agent so that Service Broker can obtain the necessary lock.

DISABLE_BROKER

Specifies that Service Broker is disabled for the specified database. Message delivery is stopped, and the is_broker_enabled flag is set to false in the sys.databases catalog view. The database retains the existing Service Broker identifier.

NEW_BROKER

Specifies that the database should receive a new broker identifier. Because the database is considered to be a new service broker, all existing conversations in the database are immediately removed without producing end dialog messages. Any route that references the old Service Broker identifier must be re-created with the new identifier.

ERROR_BROKER_CONVERSATIONS

Specifies that Service Broker message delivery is enabled. This preserves the existing Service Broker identifier for the database. Service Broker ends all conversations in the database with an error. This enables applications to perform regular cleanup for existing conversations.

HONOR_BROKER_PRIORITY {ON | OFF}

ON

Send operations take into consideration the priority levels that are assigned to conversations. Messages from conversations that have high priority levels are sent before messages from conversations that are assigned low priority levels.

OFF

Send operations run as if all conversations have the default priority level.

Changes to the HONOR_BROKER_PRIORITY option take effect immediately for new dialogs or dialogs that have no messages waiting to be sent. Dialogs that have messages waiting to be sent when ALTER DATABASE is run will not pick up the new setting until some of the messages for the dialog have been sent. The amount of time before all dialogs start using the new setting can vary considerably.

The current setting of this property is reported in the is_broker_priority_honored column in the sys.databases catalog view.

<snapshot_option> ::=

Determines the transaction isolation level.

ALLOW_SNAPSHOT_ISOLATION { ON | OFF }

ON

Enables Snapshot option at the database level. When it is enabled, DML statements start generating row versions even when no transaction uses Snapshot Isolation. Once this option is enabled, transactions can specify the SNAPSHOT transaction isolation level. When a transaction runs at the SNAPSHOT isolation level, all statements see a snapshot of data as it exists at the start of the transaction. If a transaction running at the SNAPSHOT isolation level accesses data in multiple databases, either ALLOW_SNAPSHOT_ISOLATION must be set to ON in all the databases, or each statement in the transaction must use locking hints on any reference in a FROM clause to a table in a database where ALLOW_SNAPSHOT_ISOLATION is OFF.

OFF

Turns off the Snapshot option at the database level. Transactions cannot specify the SNAPSHOT transaction isolation level.

When you set ALLOW_SNAPSHOT_ISOLATION to a new state (from ON to OFF, or from OFF to ON), ALTER DATABASE does not return control to the caller until all existing transactions in the database are committed. If the database is already in the state specified in the ALTER DATABASE statement, control is returned to the caller immediately. If the ALTER DATABASE statement does not return quickly, use sys.dm_tran_active_snapshot_database_transactions to determine whether there are long-running transactions. If the ALTER DATABASE statement is canceled, the database remains in the state it was in when ALTER DATABASE was started. The sys.databases catalog view indicates the state of snapshot-isolation transactions in the database. If **snapshot_isolation_state_desc** = IN_TRANSITION_TO_ON, ALTER DATABASE ALLOW_SNAPSHOT_ISOLATION OFF will pause six seconds and retry the operation.

You cannot change the state of ALLOW_SNAPSHOT_ISOLATION if the database is OFFLINE.

If you set ALLOW_SNAPSHOT_ISOLATION in a READ_ONLY database, the setting will be retained if the database is later set to READ_WRITE.

You can change the ALLOW_SNAPSHOT_ISOLATION settings for the master, model, msdb, and tempdb databases. If you change the setting for tempdb, the setting is retained every time the instance of the Database Engine is stopped and restarted. If you change the setting for model, that setting becomes the default for any new databases that are created, except for tempdb.

The option is ON, by default, for the master and msdb databases.

The current setting of this option can be determined by examining the snapshot_isolation_state column in the sys.databases catalog view.

READ_COMMITTED_SNAPSHOT { ON | OFF }

ON

Enables Read-Committed Snapshot option at the database level. When it is enabled, DML statements start generating row versions even when no transaction uses Snapshot

Isolation. Once this option is enabled, the transactions specifying the read committed isolation level use row versioning instead of locking. When a transaction runs at the read committed isolation level, all statements see a snapshot of data as it exists at the start of the statement.

OFF

Turns off Read-Committed Snapshot option at the database level. Transactions specifying the READ COMMITTED isolation level use locking.

To set READ_COMMITTED_SNAPSHOT ON or OFF, there must be no active connections to the database except for the connection executing the ALTER DATABASE command. However, the database does not have to be in single-user mode. You cannot change the state of this option when the database is OFFLINE.

If you set READ_COMMITTED_SNAPSHOT in a READ_ONLY database, the setting will be retained when the database is later set to READ_WRITE.

READ_COMMITTED_SNAPSHOT cannot be turned ON for the master, tempdb, or msdb system databases. If you change the setting for model, that setting becomes the default for any new databases created, except for tempdb.

The current setting of this option can be determined by examining the is_read_committed_snapshot_on column in the sys.databases catalog view.

<sql_option> ::=

Controls the ANSI compliance options at the database level.

ANSI_NULL_DEFAULT { ON | OFF }

Determines the default value, NULL or NOT NULL, of a column or CLR user-defined type for which the nullability is not explicitly defined in CREATE TABLE or ALTER TABLE statements. Columns that are defined with constraints follow constraint rules regardless of this setting.

ON

The default value is NULL.

OFF

The default value is NOT NULL.

Connection-level settings that are set by using the SET statement override the default database-level setting for ANSI_NULL_DEFAULT. By default, ODBC and OLE DB clients issue a connection-level SET statement setting ANSI_NULL_DEFAULT to ON for the session when connecting to an instance of SQL Server. For more information, see SET ANSI_NULL_DFLT_ON.

For ANSI compatibility, setting the database option ANSI_NULL_DEFAULT to ON changes the database default to NULL.

The status of this option can be determined by examining the is_ansi_null_default_on column in the sys.databases catalog view or the IsAnsiNullDefault property of the DATABASEPROPERTYEX function.

ANSI_NULLS { ON | OFF }

ON

All comparisons to a null value evaluate to UNKNOWN.

OFF

Comparisons of non-UNICODE values to a null value evaluate to TRUE if both values are NULL.

Important

In a future version of SQL Server, ANSI_NULLS will always be ON and any applications that explicitly set the option to OFF will produce an error. Avoid using this feature in new development work, and plan to modify applications that currently use this feature.

Connection-level settings that are set by using the SET statement override the default database setting for ANSI_NULLS. By default, ODBC and OLE DB clients issue a connection-level SET statement setting ANSI_NULLS to ON for the session when connecting to an instance of SQL Server. For more information, see SET ANSI_NULLS.

SET ANSI_NULLS also must be set to ON when you create or make changes to indexes on computed columns or indexed views.

The status of this option can be determined by examining the is_ansi_nulls_on column in the sys.databases catalog view or the IsAnsiNullsEnabled property of the DATABASEPROPERTYEX function.

ANSI_PADDING { ON | OFF }

ON

Strings are padded to the same length before conversion or inserting to a **varchar** or **nvarchar** data type.

Trailing blanks in character values inserted into **varchar** or **nvarchar** columns and trailing zeros in binary values inserted into **varbinary** columns are not trimmed. Values are not padded to the length of the column.

OFF

Trailing blanks for **varchar** or **nvarchar** and zeros for **varbinary** are trimmed.

When OFF is specified, this setting affects only the definition of new columns.

Important

In a future version of SQL Server, ANSI_PADDING will always be ON and any applications that explicitly set the option to OFF will produce an error. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. We recommend that you always set ANSI_PADDING to ON. ANSI_PADDING must be ON when you create or manipulate indexes on computed columns or indexed views.

char(*n*) and **binary(*n*)** columns that allow for nulls are padded to the length of the column when ANSI_PADDING is set to ON, but trailing blanks and zeros are trimmed when

ANSI_PADDING is OFF. **char(*n*)** and **binary(*n*)** columns that do not allow nulls are always padded to the length of the column.

Connection-level settings that are set by using the SET statement override the default database-level setting for ANSI_PADDING. By default, ODBC and OLE DB clients issue a connection-level SET statement setting ANSI_PADDING to ON for the session when connecting to an instance of SQL Server. For more information, see SET ANSI_PADDING.

Important

The status of this option can be determined by examining the is_ansi_padding_on column in the sys.databases catalog view or the IsAnsiPaddingEnabled property of the DATABASEPROPERTYEX function.

ANSI_WARNINGS { ON | OFF }

ON

Errors or warnings are issued when conditions such as divide-by-zero occur or null values appear in aggregate functions.

OFF

No warnings are raised and null values are returned when conditions such as divide-by-zero occur.

SET ANSI_WARNINGS must be set to ON when you create or make changes to indexes on computed columns or indexed views.

Connection-level settings that are set by using the SET statement override the default database setting for ANSI_WARNINGS. By default, ODBC and OLE DB clients issue a connection-level SET statement setting ANSI_WARNINGS to ON for the session when connecting to an instance of SQL Server. For more information, see SET ANSI_WARNINGS.

The status of this option can be determined by examining the is_ansi_warnings_on column in the sys.databases catalog view or the IsAnsiWarningsEnabled property of the DATABASEPROPERTYEX function.

ARITHABORT { ON | OFF }

ON

A query is ended when an overflow or divide-by-zero error occurs during query execution.

OFF

A warning message is displayed when one of these errors occurs, but the query, batch, or transaction continues to process as if no error occurred.

SET ARITHABORT must be set to ON when you create or make changes to indexes on computed columns or indexed views.

The status of this option can be determined by examining the is_arithabort_on column in the

sys.databases catalog view or the IsArithmeticAbortEnabled property of the DATABASEPROPERTYEX function.

COMPATIBILITY_LEVEL { 90 | 100 | 110 }

For more information, see ALTER DATABASE Compatibility Level (Transact-SQL).

CONCAT_NULL_YIELDS_NULL { ON | OFF }

ON

The result of a concatenation operation is NULL when either operand is NULL. For example, concatenating the character string "This is" and NULL causes the value NULL, instead of the value "This is".

OFF

The null value is treated as an empty character string.

CONCAT_NULL_YIELDS_NULL must be set to ON when you create or make changes to indexes on computed columns or indexed views.

Important

In a future version of SQL Server, CONCAT_NULL_YIELDS_NULL will always be ON and any applications that explicitly set the option to OFF will produce an error. Avoid using this feature in new development work, and plan to modify applications that currently use this feature.

Connection-level settings that are set by using the SET statement override the default database setting for CONCAT_NULL_YIELDS_NULL. By default, ODBC and OLE DB clients issue a connection-level SET statement setting CONCAT_NULL_YIELDS_NULL to ON for the session when connecting to an instance of SQL Server. For more information, see SET CONCAT_NULL_YIELDS_NULL.

The status of this option can be determined by examining the is_concat_null_yields_null_on column in the sys.databases catalog view or the IsNullConcat property of the DATABASEPROPERTYEX function.

QUOTED_IDENTIFIER { ON | OFF }

ON

Double quotation marks can be used to enclose delimited identifiers.

All strings delimited by double quotation marks are interpreted as object identifiers. Quoted identifiers do not have to follow the Transact-SQL rules for identifiers. They can be keywords and can include characters not generally allowed in Transact-SQL identifiers. If a single quotation mark (') is part of the literal string, it can be represented by double quotation marks (").

OFF

Identifiers cannot be in quotation marks and must follow all Transact-SQL rules for identifiers. Literals can be delimited by either single or double quotation marks.

SQL Server also allows for identifiers to be delimited by square brackets ([ ]). Bracketed identifiers can always be used, regardless of the setting of QUOTED_IDENTIFIER. For more information, see Database Identifiers.

When a table is created, the QUOTED IDENTIFIER option is always stored as ON in the metadata of the table, even if the option is set to OFF when the table is created.

Connection-level settings that are set by using the SET statement override the default

database setting for QUOTED_IDENTIFIER. By default, ODBC and OLE DB clients issue a connection-level SET statement setting QUOTED_IDENTIFIER to ON when connecting to an instance of SQL Server. For more information, see SET QUOTED_IDENTIFIER.

The status of this option can be determined by examining the is_quoted_identifier_on column in the sys.databases catalog view or the IsQuotedIdentifiersEnabled property of the DATABASEPROPERTYEX function.

NUMERIC_ROUNDABORT { ON | OFF }

ON

An error is generated when loss of precision occurs in an expression.

OFF

Losses of precision do not generate error messages and the result is rounded to the precision of the column or variable storing the result.

NUMERIC_ROUNDABORT must be set to OFF when you create or make changes to indexes on computed columns or indexed views.

The status of this option can be determined by examining the is_numeric_roundabort_on column in the sys.databases catalog view or the IsNumericRoundAbortEnabled property of the DATABASEPROPERTYEX function.

RECURSIVE_TRIGGERS { ON | OFF }

ON

Recursive firing of AFTER triggers is allowed.

OFF

Only direct recursive firing of AFTER triggers is not allowed. To also disable indirect recursion of AFTER triggers, set the nested triggers server option to **0** by using **sp_configure**.

Note

Only direct recursion is prevented when RECURSIVE_TRIGGERS is set to OFF. To disable indirect recursion, you must also set the nested triggers server option to 0.

The status of this option can be determined by examining the is_recursive_triggers_on column in the sys.databases catalog view or the IsRecursiveTriggersEnabled property of the DATABASEPROPERTYEX function.

WITH <termination> ::=

Specifies when to roll back incomplete transactions when the database is transitioned from one state to another. If the termination clause is omitted, the ALTER DATABASE statement waits indefinitely if there is any lock on the database. Only one termination clause can be specified, and it follows the SET clauses.

Note

Not all database options use the WITH <termination> clause. For more information, see the table under "Setting Options of the "Remarks" section of this topic.

ROLLBACK AFTER integer [SECONDS] | ROLLBACK IMMEDIATE

Specifies whether to roll back after the specified number of seconds or immediately.

NO_WAIT

Specifies that if the requested database state or option change cannot complete immediately without waiting for transactions to commit or roll back on their own, the request will fail.

Remarks

Setting Options

To retrieve current settings for database options, use the sys.databases catalog view or DATABASEPROPERTYEX

After you set a database option, the modification takes effect immediately.

To change the default values for any one of the database options for all newly created databases, change the appropriate database option in the model database.

Not all database options use the WITH <termination> clause or can be specified in combination with other options. The following table lists these options and their option and termination status.

Options category	Can be specified with other options	Can use the WITH <termination> clause
<db_state_option>	Yes	Yes
<db_user_access_option>	Yes	Yes
<db_update_option>	Yes	Yes
<external_access_option>	Yes	No
<cursor_option>	Yes	No
<auto_option>	Yes	No
<sql_option>	Yes	No
<recovery_option>	Yes	No
<target_recovery_time_option>	No	Yes
<database_mirroring_option>	No	No
ALLOW_SNAPSHOT_ISOLATION	No	No
READ_COMMITTED_SNAPSHOT	No	Yes
<service_broker_option>	Yes	No

Options category	Can be specified with other options	Can use the WITH <termination> clause
DATE_CORRELATION_OPTIMIZATION	Yes	Yes
<parameterization_option>	Yes	Yes
<change_tracking_option>	Yes	Yes
<db_encryption>	Yes	No

The plan cache for the instance of SQL Server is cleared by setting one of the following options:

OFFLINE	READ_WRITE
ONLINE	MODIFY FILEGROUP DEFAULT
MODIFY_NAME	MODIFY FILEGROUP READ_WRITE
COLLATE	MODIFY FILEGROUP READ_ONLY
READ_ONLY	

Clearing the plan cache causes a recompilation of all subsequent execution plans and can cause a sudden, temporary decrease in query performance. For each cleared cachestore in the plan cache, the SQL Server error log contains the following informational message: "SQL Server has encountered %d occurrence(s) of cachestore flush for the '%s' cachestore (part of plan cache) due to some database maintenance or reconfigure operations". This message is logged every five minutes as long as the cache is flushed within that time interval.

Examples

A. Setting options on a database

The following example sets the recovery model and data page verification options for the AdventureWorks2012 sample database.

```
USE master;
GO
ALTER DATABASE AdventureWorks2012
SET RECOVERY FULL, PAGE_VERIFY CHECKSUM;
GO
```

B. Setting the database to READ_ONLY

Changing the state of a database or filegroup to READ_ONLY or READ_WRITE requires exclusive access to the database. The following example sets the database to `SINGLE_USER` mode to

obtain exclusive access. The example then sets the state of the AdventureWorks2012 database to `READ_ONLY` and returns access to the database to all users.

Note
This example uses the termination option `WITH ROLLBACK IMMEDIATE` in the first `ALTER DATABASE` statement. All incomplete transactions will be rolled back and any other connections to the database will be immediately disconnected.

```
USE master;
GO
ALTER DATABASE AdventureWorks2012
SET SINGLE_USER
WITH ROLLBACK IMMEDIATE;
GO
ALTER DATABASE AdventureWorks2012
SET READ_ONLY;
GO
ALTER DATABASE AdventureWorks2012
SET MULTI_USER;
GO
```

C. Enabling snapshot isolation on a database

The following example enables the snapshot isolation framework option for the AdventureWorks2012 database.

```
USE AdventureWorks2012;
GO
-- Check the state of the snapshot_isolation_framework
-- in the database.
SELECT name, snapshot_isolation_state,
     snapshot_isolation_state_desc AS description
FROM sys.databases
WHERE name = N'AdventureWorks2012';
GO
USE master;
GO
ALTER DATABASE AdventureWorks2012
    SET ALLOW_SNAPSHOT_ISOLATION ON;
GO
```

```
-- Check again.
SELECT name, snapshot_isolation_state,
     snapshot_isolation_state_desc AS description
FROM sys.databases
WHERE name = N'AdventureWorks2012';
GO
```

The result set shows that the snapshot isolation framework is enabled.

```
name                 snapshot_isolation_state  description
-------------------- ------------------------  ----------
AdventureWorks2012 1                           ON
```

D. Enabling, modifying, and disabling change tracking

The following example enables change tracking for the AdventureWorks2012 database and sets the retention period to `4` days.

```
ALTER DATABASE AdventureWorks2012
SET CHANGE_TRACKING = ON
(AUTO_CLEANUP = ON, CHANGE_RETENTION = 2 DAYS);
```

The following example shows how to change the retention period to `3` days.

```
ALTER DATABASE AdventureWorks2012
SET CHANGE_TRACKING (CHANGE_RETENTION = 3 DAYS);
```

The following example shows how to disable change tracking for the AdventureWorks2012 database.

```
ALTER DATABASE AdventureWorks2012
SET CHANGE_TRACKING = OFF;
```

See Also

ALTER DATABASE Compatibility Level (Transact-SQL)
ALTER DATABASE Database Mirroring (Transact-SQL)
ALTER DATABASE SET HADR (Transact-SQL)
Using Statistics to Improve Query Performance
CREATE DATABASE
Configuring and Managing Change Tracking
DATABASEPROPERTYEX

DROP DATABASE

SET TRANSACTION ISOLATION LEVEL

sp_configure

sys.databases (Transact-SQL)

sys.data_spaces (Transact-SQL)

ALTER DATABASE Database Mirroring

Note

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

Controls database mirroring for a database. Values specified with the database mirroring options apply to both copies of the database and to the database mirroring session as a whole. Only one <database_mirroring_option> is permitted per ALTER DATABASE statement.

Note

We recommend that you configure database mirroring during off-peak hours because configuration can affect performance.

For ALTER DATABASE options, see ALTER DATABASE (Transact-SQL). For ALTER DATABASE SET options, see ALTER DATABASE SET Options (Transact-SQL).

Transact-SQL Syntax Conventions

Syntax

```
ALTER DATABASE database_name
SET { <partner_option> | <witness_option> }
  <partner_option> ::=
   PARTNER { = 'partner_server'
         | FAILOVER
         | FORCE_SERVICE_ALLOW_DATA_LOSS
         | OFF
         | RESUME
         | SAFETY { FULL | OFF }
         | SUSPEND
         | TIMEOUT integer
         }
  <witness_option> ::=
   WITNESS { = 'witness_server'
```

| OFF
}

Arguments

Important
A SET PARTNER or SET WITNESS command can complete successfully when entered, but fail later.

Note
ALTER DATABASE database mirroring options are not available for a contained database.

database_name

Is the name of the database to be modified.

PARTNER <partner_option>

Controls the database properties that define the failover partners of a database mirroring session and their behavior. Some SET PARTNER options can be set on either partner; others are restricted to the principal server or to the mirror server. For more information, see the individual PARTNER options that follow. A SET PARTNER clause affects both copies of the database, regardless of the partner on which it is specified.

To execute a SET PARTNER statement, the STATE of the endpoints of both partners must be set to STARTED. Note, also, that the ROLE of the database mirroring endpoint of each partner server instance must be set to either PARTNER or ALL. For information about how to specify an endpoint, see How to: Create a Mirroring Endpoint for Windows Authentication (Transact-SQL). To learn the role and state of the database mirroring endpoint of a server instance, on that instance, use the following Transact-SQL statement:

```
SELECT role_desc, state_desc FROM
sys.database_mirroring_endpoints
```

<partner_option> ::=

Note
Only one <partner_option> is permitted per SET PARTNER clause.

'partner_server'

Specifies the server network address of an instance of SQL Server to act as a failover partner in a new database mirroring session. Each session requires two partners: one starts as the principal server, and the other starts as the mirror server. We recommend that these partners reside on different computers.

This option is specified one time per session on each partner. Initiating a database mirroring session requires two ALTER DATABASE database SET PARTNER = 'partner_server' statements. Their order is significant. First, connect to the mirror server, and specify the principal server instance as partner_server (SET PARTNER = 'principal_server'). Second, connect to the principal server, and specify the mirror server instance as partner_server

(SET PARTNER = 'mirror_server'); this starts a database mirroring session between these two partners. For more information, see Overview of Setting Up Database Mirroring (Transact-SQL).

The value of partner_server is a server network address. This has the following syntax:

TCP**://***<system-address>***:***<port>*

where

- *<system-address>* is a string, such as a system name, a fully qualified domain name, or an IP address, that unambiguously identifies the destination computer system.
- *<port>* is a port number that is associated with the mirroring endpoint of the partner server instance.

For more information, see Specifying a Server Network Address (Database Mirroring).

The following example illustrates the SET PARTNER = 'partner_server' clause:

```
'TCP://MYSERVER.mydomain.Adventure-Works.com:7777'
```

Important

If a session is set up by using the ALTER DATABASE statement instead of SQL Server Management Studio, the session is set to full transaction safety by default (SAFETY is set to FULL) and runs in high-safety mode without automatic failover. To allow automatic failover, configure a witness; to run in high-performance mode, turn off transaction safety (SAFETY OFF).

FAILOVER

Manually fails over the principal server to the mirror server. You can specify FAILOVER only on the principal server. This option is valid only when the SAFETY setting is FULL (the default).

The FAILOVER option requires **master** as the database context.

FORCE_SERVICE_ALLOW_DATA_LOSS

Forces database service to the mirror database after the principal server fails with the database in an unsynchronized state or in a synchronized state when automatic failover does not occur.

We strongly recommend that you force service only if the principal server is no longer running. Otherwise, some clients might continue to access the original principal database instead of the new principal database.

FORCE_SERVICE_ALLOW_DATA_LOSS is available only on the mirror server and only under all the following conditions:

- The principal server is down.
- WITNESS is set to OFF or the witness is connected to the mirror server.

Force service only if you are willing to risk losing some data in order to restore service to the database immediately.

Forcing service suspends the session, temporarily preserving all the data in the original

principal database. Once the original principal is in service and able to communicate with the new principal server, the database administrator can resume service. When the session resumes, any unsent log records and the corresponding updates are lost.

OFF

Removes a database mirroring session and removes mirroring from the database. You can specify OFF on either partner. For information, see about the impact of removing mirroring, see Removing Database Mirroring.

RESUME

Resumes a suspended database mirroring session. You can specify RESUME only on the principal server.

SAFETY { FULL | OFF }

Sets the level of transaction safety. You can specify SAFETY only on the principal server.

The default is FULL. With full safety, the database mirroring session runs synchronously (in *high-safety mode*). If SAFETY is set to OFF, the database mirroring session runs asynchronously (in *high-performance mode*).

The behavior of high-safety mode depends partly on the witness, as follows:

- When safety is set to FULL and a witness is set for the session, the session runs in high-safety mode with automatic failover. When the principal server is lost, the session automatically fails over if the database is synchronized and the mirror server instance and witness are still connected to each other (that is, they have quorum). For more information, see Quorum in Database Mirroring Sessions.

  If a witness is set for the session but is currently disconnected, the loss of the mirror server causes the principal server to go down.

- When safety is set to FULL and the witness is set to OFF, the session runs in high-safety mode without automatic failover. If the mirror server instance goes down, the principal server instance is unaffected. If the principal server instance goes down, you can force service (with possible data loss) to the mirror server instance.

If SAFETY is set to OFF, the session runs in high-performance mode, and automatic failover and manual failover are not supported. However, problems on the mirror do not affect the principal, and if the principal server instance goes down, you can, if necessary, force service (with possible data loss) to the mirror server instance—if WITNESS is set to OFF or the witness is currently connected to the mirror. For more information on forcing service, see "FORCE_SERVICE_ALLOW_DATA_LOSS" earlier in this section.

Important

High-performance mode is not intended to use a witness. However, whenever you set SAFETY to OFF, we strongly recommend that you ensure that WITNESS is set to OFF.

SUSPEND

Pauses a database mirroring session.

You can specify SUSPEND on either partner.

TIMEOUT integer

Specifies the time-out period in seconds. The time-out period is the maximum time that a server instance waits to receive a PING message from another instance in the mirroring session before considering that other instance to be disconnected.

You can specify the TIMEOUT option only on the principal server. If you do not specify this option, by default, the time period is 10 seconds. If you specify 5 or greater, the time-out period is set to the specified number of seconds. If you specify a time-out value of 0 to 4 seconds, the time-out period is automatically set to 5 seconds.

Important

We recommend that you keep the time-out period at 10 seconds or greater. Setting the value to less than 10 seconds creates the possibility of a heavily loaded system missing PINGs and declaring a false failure.

For more information, see Possible Failures During Database Mirroring Sessions.

WITNESS <witness_option>

Controls the database properties that define a database mirroring witness. A SET WITNESS clause affects both copies of the database, but you can specify SET WITNESS only on the principal server. If a witness is set for a session, quorum is required to serve the database, regardless of the SAFETY setting; for more information, see Quorum in Database Mirroring Sessions.

We recommend that the witness and failover partners reside on separate computers. For information about the witness, see The Role of the Witness.

To execute a SET WITNESS statement, the STATE of the endpoints of both the principal and witness server instances must be set to STARTED. Note, also, that the ROLE of the database mirroring endpoint of a witness server instance must be set to either WITNESS or ALL. For information about specifying an endpoint, see The Database Mirroring Endpoint.

To learn the role and state of the database mirroring endpoint of a server instance, on that instance, use the following Transact-SQL statement:

```
SELECT role_desc, state_desc FROM
sys.database_mirroring_endpoints
```

Note

Database properties cannot be set on the witness.

<witness_option> ::=

Note

Only one <witness_option> is permitted per SET WITNESS clause.

'witness_server'

Specifies an instance of the Database Engine to act as the witness server for a database mirroring session. You can specify SET WITNESS statements only on the principal server.

In a SET WITNESS = 'witness_server' statement, the syntax of *witness_server* is the same as the syntax of *partner_server*.

OFF

Removes the witness from a database mirroring session. Setting the witness to OFF disables automatic failover. If the database is set to FULL SAFETY and the witness is set to OFF, a failure on the mirror server causes the principal server to make the database unavailable.

Remarks

Examples

A. Creating a database mirroring session with a witness

Setting up database mirroring with a witness requires configuring security and preparing the mirror database, and also using ALTER DATABASE to set the partners. For an example of the complete setup process, see Setting Up Database Mirroring .

B. Manually failing over a database mirroring session

Manual failover can be initiated from either database mirroring partner. Before failing over, you should verify that the server you believe to be the current principal server actually is the principal server. For example, for the database, on that server instance that you think is the current principal server, execute the following query:

```
SELECT db.name, m.mirroring_role_desc
FROM sys.database_mirroring m
JOIN sys.databases db
ON db.database_id = m.database_id
WHERE db.name = N'AdventureWorks2012';
GO
```

If the server instance is in fact the principal, the value of `mirroring_role_desc` is `Principal`. If this server instance were the mirror server, the `SELECT` statement would return `Mirror`.

The following example assumes that the server is the current principal.

1. Manually fail over to the database mirroring partner:

   ```
   ALTER DATABASE AdventureWorks2012 SET PARTNER FAILOVER;
   GO
   ```

2. To verify the results of the failover on the new mirror, execute the following query:

   ```
   SELECT db.name, m.mirroring_role_desc
   FROM sys.database_mirroring m
   JOIN sys.databases db
   ```

```
    ON db.database_id = m.database_id
    WHERE db.name = N'AdventureWorks2012';
    GO
```

The current value of `mirroring_role_desc` is now `Mirror`.

See Also

CREATE DATABASE

DATABASEPROPERTYEX

sys.database_mirroring_witnesses

ALTER DATABASE SET HADR

This topic contains the ALTER DATABASE syntax for setting AlwaysOn Availability Groups options on a secondary database. Only one SET HADR option is permitted per ALTER DATABASE statement. These options are supported only on secondary replicas.

Transact-SQL Syntax Conventions

Syntax

```
ALTER DATABASE database_name
  SET HADR
  {
     { AVAILABILITY GROUP = group_name | OFF }
  | { SUSPEND | RESUME }
  }
;
```

Arguments

database_name

Is the name of the secondary database to be modified.

SET HADR

Executes the specified AlwaysOn Availability Groups command on the specified database.

{ AVAILABILITY GROUP = group_name | OFF }

Joins or removes the availability database from the specified availability group, as follows:

group_name

Joins the specified database on the secondary replica that is hosted by the server instance on which you execute the command to the availability group specified by group_name.

The prerequisites for this operation are as follows:

- The database must already have been added to the availability group on the primary

replica.

- The primary replica must be active. For information about how troubleshoot an inactive primary replica, see Troubleshooting AlwaysOn Availability Groups Configuration (SQL Server).
- The primary replica must be online, and the secondary replica must be connected to the primary replica.
- The secondary database must have been restored using WITH NORECOVERY from recent database and log backups of the primary database, ending with a log backup that is recent enough to permit the secondary database to catch up to the primary database.

Note

To add a database to the availability group, connect to the server instance that hosts the primary replica, and use the ALTER AVAILABILITY GROUP *group_name* ADD DATABASE *database_name* statement.

For more information, see Joining a Secondary Database to an Availability Group (SQL Server).

OFF

Removes the specified secondary database from the availability group.

Removing a secondary database can be useful if it has fallen far behind the primary database, and you do not want to wait for the secondary database to catch up. After removing the secondary database, you can update it by restoring a sequence of backups ending with a recent log backup (using RESTORE … WITH NORECOVERY).

Important

To completely remove an availability database from an availability group, connect to the server instance that hosts the primary replica, and use the ALTER AVAILABILITY GROUP *group_name* REMOVE DATABASE *availability_database_name* statement. For more information, see Removing an Availability Database from an Availability Group (SQL Server).

SUSPEND

Suspends data movement on a secondary database. A SUSPEND command returns as soon as it has been accepted by the replica that hosts the target database, but actually suspending the database occurs asynchronously.

The scope of the impact depends on where you execute the ALTER DATABASE statement:

- If you suspend a secondary database on a secondary replica, only the local secondary database is suspended. Existing connections on the readable secondary remain usable. New connections to the suspended database on the readable secondary are not allowed until data movement is resumed.
- If you suspend a database on the primary replica, data movement is suspended to the corresponding secondary databases on every secondary replica. Existing connections on

a readable secondary remain usable and new connections can be made.

- When data movement is suspended due to a forced manual failover, connections to the new secondary replica are not allowed while data movement is suspended.

When a database on a secondary replica is suspended, both the database and replica become unsynchronized and are marked as NOT SYNCHRONIZED.

Important

While a secondary database is suspended, the send queue of the corresponding primary database will accumulate unsent transaction log records. Connections to the secondary replica return data that was available at the time the data movement was suspended.

Note

Suspending and resuming an AlwaysOn secondary database does not directly affect the availability of the primary database, though suspending a secondary database can impact redundancy and failover capabilities for the primary database, until the suspended secondary database is resumed. This is in contrast to database mirroring, where the mirroring state is suspended on both the mirror database and the principal database until mirroring is resumed. Suspending an AlwaysOn primary database suspends data movement on all the corresponding secondary databases, and redundancy and failover capabilities cease for that database until the primary database is resumed.

For more information, see Suspend a Secondary Database in an Availability Group (SQL Server).

RESUME

Resumes suspended data movement on the specified secondary database. A RESUME command returns as soon as it has been accepted by the replica that hosts the target database, but actually resuming the database occurs asynchronously.

The scope of the impact depends on where you execute the ALTER DATABASE statement:

- If you resume a secondary database on a secondary replica, only the local secondary database is resumed. Data movement is resumed unless the database has also been suspended on the primary replica.
- If you resume a database on the primary replica, data movement is resumed to every secondary replica on which the corresponding secondary database has not also been suspended locally. To resume a secondary database that was individually suspended on a secondary replica, connect to the server instance that hosts the secondary replica and resume the database there.

  Under synchronous-commit mode, the database state changes to SYNCHRONIZING. If no other database is currently suspended, the replica state also changes to SYNCHRONIZING.

  For more information, see Resume a Secondary Database in an Availability Group (SQL Server).

Database States

When a secondary database is joined to an availability group, the local secondary replica changes the state of that secondary database from RESTORING to ONLINE. If a secondary database is removed from the availability group, it is set back to the RESTORING state by the local secondary replica. This allows you to apply subsequent log backups from the primary database to that secondary database.

Restrictions

Execute ALTER DATABASE statements outside of both transactions and batches.

Security

Permissions

Requires ALTER permission on the database. Joining a database to an availability group requires membership in the **db_owner** fixed database role.

Examples

The following example joins the secondary database, `AccountsDb1`, to the local secondary replica of the `AccountsAG` availability group.

```
ALTER DATABASE AccountsDb1 SET HADR AVAILABILITY GROUP = AccountsAG;
```

Note
To see this Transact-SQL statement used in context, see Example: Setting Up an Availability Group Using Windows Authentication (Transact-SQL).

See Also

ALTER DATABASE (Transact-SQL)

ALTER AVAILABILITY GROUP (Transact-SQL)

CREATE AVAILABILITY GROUP (Transact-SQL)

Overview of AlwaysOn Availability Groups (SQL Server)

Troubleshoot AlwaysOn Availability Groups Configuration (SQL Server)

ALTER DATABASE Compatibility Level

Sets certain database behaviors to be compatible with the specified version of SQL Server. For other ALTER DATABASE options, see ALTER DATABASE (Transact-SQL).

Transact-SQL Syntax Conventions

Syntax

```
ALTER DATABASE database_name
SET COMPATIBILITY_LEVEL = { 90 | 100 | 110 }
```

Arguments

database_name

Is the name of the database to be modified.

COMPATIBILITY_LEVEL { 90 | 100 | 110 }

Is the version of SQL Server with which the database is to be made compatible. The value must be one of the following:

90 = SQL Server 2005

100 = SQL Server 2008 and SQL Server 2008 R2

110 = SQL Server 2012

Remarks

For all installations of SQL Server 2012, the default compatibility level is 110. Databases created in SQL Server 2012 are set to this level unless the **model** database has a lower compatibility level. When a database is upgraded to SQL Server 2012 from any earlier version of SQL Server, the database retains its existing compatibility level if it is at least 90. Upgrading a database with a compatibility level below 90 sets the database to compatibility level 90. This applies to both system and user databases. Use **ALTER DATABASE** to change the compatibility level of the database. To view the current compatibility level of a database, query the **compatibility_level** column in the **sys.databases** catalog view.

Using Compatibility Level for Backward Compatibility

Compatibility level affects behaviors only for the specified database, not for the entire server. Compatibility level provides only partial backward compatibility with earlier versions of SQL Server. Use compatibility level as an interim migration aid to work around version differences in the behaviors that are controlled by the relevant compatibility-level setting. If existing SQL Server applications are affected by behavioral differences in SQL Server 2012, convert the application to work properly. Then use **ALTER DATABASE** to change the compatibility level to 100. The new compatibility setting for a database takes effect when the database is next made current (whether as the default database on login or on being specified in a USE statement).

Best Practices

Changing the compatibility level while users are connected to the database can produce incorrect result sets for active queries. For example, if the compatibility level changes while a query plan is being compiled, the compiled plan might be based on both the old and new compatibility levels, resulting in an incorrect plan and potentially inaccurate results. Furthermore, the problem may be compounded if the plan is placed in the plan cache and reused for subsequent queries. To avoid inaccurate query results, we recommend the following procedure to change the compatibility level of a database:

1. Set the database to single-user access mode by using ALTER DATABASE SET SINGLE_USER.
2. Change the compatibility level of the database.
3. Put the database in multiuser access mode by using ALTER DATABASE SET MULTI_USER.
4. For more information about setting the access mode of a database, see ALTER DATABASE (Transact-SQL).

Compatibility Levels and Stored Procedures

When a stored procedure executes, it uses the current compatibility level of the database in which it is defined. When the compatibility setting of a database is changed, all of its stored procedures are automatically recompiled accordingly.

Differences Between Compatibility Level 90 and Level 100

This section describes new behaviors introduced with compatibility level 100.

Compatibility-level setting of 90	Compatibility-level setting of 100	Possibility of impact
The QUOTED_IDENTIFER setting is always set to ON for multistatement table-valued functions when they are created regardless of the session level setting.	The QUOTED IDENTIFIER session setting is honored when multistatement table-valued functions are created.	Medium
When you create or alter a partition function, **datetime** and **smalldatetime** literals in the function are evaluated assuming US_English as the language setting.	The current language setting is used to evaluate **datetime** and **smalldatetime** literals in the partition function.	Medium
The FOR BROWSE clause is allowed (and ignored) in INSERT and SELECT INTO statements.	The FOR BROWSE clause is not allowed in INSERT and SELECT INTO statements.	Medium
Full-text predicates are allowed in the OUTPUT clause.	Full-text predicates are not allowed in the OUTPUT clause.	Low
CREATE FULLTEXT STOPLIST, ALTER FULLTEXT STOPLIST, and DROP FULLTEXT STOPLIST are not supported. The system stoplist is automatically associated with new full-text indexes.	CREATE FULLTEXT STOPLIST, ALTER FULLTEXT STOPLIST, and DROP FULLTEXT STOPLIST are supported.	Low
MERGE is not enforced as a reserved keyword.	MERGE is a fully reserved keyword. The MERGE statement is supported under both 100 and 90 compatibility levels.	Low
Using the <dml_table_source> argument of the INSERT statement raises a syntax error.	You can capture the results of an OUTPUT clause in a nested INSERT, UPDATE, DELETE, or MERGE statement, and insert those results into a target table or view. This is done using the <dml_table_source> argument of the INSERT statement.	Low

Compatibility-level setting of 90	Compatibility-level setting of 100	Possibility of impact
Unless NOINDEX is specified, DBCC CHECKDB or DBCC CHECKTABLE performs both physical and logical consistency checks on a single table or indexed view and on all its nonclustered and XML indexes. Spatial indexes are not supported.	Unless NOINDEX is specified, DBCC CHECKDB or DBCC CHECKTABLE performs both physical and logical consistency checks on a single table and on all its nonclustered indexes. However, on XML indexes, spatial indexes, and indexed views, only physical consistency checks are performed by default. If WITH EXTENDED_LOGICAL_CHECKS is specified, logical checks are performed on indexed views, XML indexes, and spatial indexes, where present. By default, physical consistency checks are performed before the logical consistency checks. If NOINDEX is also specified, only the logical checks are performed.	Low
When an OUTPUT clause is used with a data manipulation language (DML) statement and a run-time error occurs during statement execution, the entire transaction is terminated and rolled back.	When an OUTPUT clause is used with a data manipulation language (DML) statement and a run-time error occurs during statement execution, the behavior depends on the SET XACT_ABORT setting. If SET XACT_ABORT is OFF, a statement abort error generated by the DML statement using the OUTPUT clause will terminate the statement, but the execution of the batch continues and the transaction is not rolled back. If SET XACT_ABORT is ON, all run-time errors generated by the DML statement using the OUTPUT clause will terminate the batch, and the transaction is rolled back.	Low
CUBE and ROLLUP are not enforced as reserved keywords.	CUBE and ROLLUP are reserved keywords within the GROUP BY clause.	Low
Strict validation is applied to elements	Lax validation is applied to elements of	Low

Compatibility-level setting of 90	Compatibility-level setting of 100	Possibility of impact
of the XML **anyType** type.	the **anyType** type. For more information, see Wildcard Components and Content Validation.	
The special attributes **xsi:nil** and **xsi:type** cannot be queried or modified by data manipulation language statements. This means that `/e/@xsi:nil` fails while `/e/@*` ignores the **xsi:nil** and **xsi:type** attributes. However, `/e` returns the **xsi:nil** and **xsi:type** attributes for consistency with `SELECT xmlCol`, even if `xsi:nil = "false"`.	The special attributes **xsi:nil** and **xsi:type** are stored as regular attributes and can be queried and modified. For example, executing the query `SELECT x.query('a/b/@*')` returns all attributes including **xsi:nil** and **xsi:type**. To exclude these types in the query, replace `@*` with `@*[namespace-uri(.) != "`*insert xsi namespace uri*`"` and not `(local-name(.) = "type"` or `local-name(.) ="nil".`	Low
A user-defined function that converts an XML constant string value to a SQL Server datetime type is marked as deterministic.	A user-defined function that converts an XML constant string value to a SQL Server datetime type is marked as non-deterministic.	Low
The XML union and list types are not fully supported.	The union and list types are fully supported including the following functionality: • Union of list • Union of union • List of atomic types • List of union	Low
The SET options required for an xQuery method are not validated when the method is contained in a view or inline table-valued function.	The SET options required for an xQuery method are validated when the method is contained in a view or inline table-valued function. An error is raised if the SET options of the method are set incorrectly.	Low
XML attribute values that contain end-of-line characters (carriage return and line feed) are not normalized according to the XML standard. That is, both characters are returned instead of	XML attribute values that contain end-of-line characters (carriage return and line feed) are normalized according to the XML standard. That is, all line breaks in external parsed entities	Low

Compatibility-level setting of 90	Compatibility-level setting of 100	Possibility of impact
a single line-feed character.	(including the document entity) are normalized on input by translating both the two-character sequence #xD #xA and any #xD that is not followed by #xA to a single #xA character. Applications that use attributes to transport string values that contain end-of-line characters will not receive these characters back as they are submitted. To avoid the normalization process, use the XML numeric character entities to encode all end-of-line characters.	
The column properties ROWGUIDCOL and IDENTITY can be incorrectly named as a constraint. For example the statement `CREATE TABLE T (C1 int CONSTRAINT MyConstraint IDENTITY)` executes, but the constraint name is not preserved and is not accessible to the user.	The column properties ROWGUIDCOL and IDENTITY cannot be named as a constraint. Error 156 is returned.	Low
Updating columns by using a two-way assignment such as `UPDATE T1 SET @v = column_name = <expression>` can produce unexpected results because the live value of the variable can be used in other clauses such as the WHERE and ON clause during statement execution instead of the statement starting value. This can cause the meanings of the predicates to change unpredictably on a per-row basis. This behavior is applicable only when the compatibility level is set to 90.	Updating columns by using a two-way assignment produces expected results because only the statement starting value of the column is accessed during statement execution.	Low
Variable assignment is allowed in a statement containing a top-level UNION operator, but returns	Variable assignment is not allowed in a statement containing a top-level UNION operator. Error 10734 is	Low

Compatibility-level setting of 90	Compatibility-level setting of 100	Possibility of impact

unexpected results. For example, in the following statements, local variable `@v` is assigned the value of the column `BusinessEntityID` from the union of two tables. By definition, when the SELECT statement returns more than one value, the variable is assigned the last value that is returned. In this case, the variable is correctly assigned the last value, however, the result set of the SELECT UNION statement is also returned.
`ALTER DATABASE AdventureWorks2012`
`SET compatibility_level = 90;`
`GO`
`USE AdventureWorks2012;`
`GO`
`DECLARE @v int;`
`SELECT @v = BusinessEntityID FROM HumanResources.Employee`
`UNION ALL`
`SELECT @v = BusinessEntityID FROM HumanResources.EmployeeAddress;`
`SELECT @v;`
	returned.
	To resolve the error, rewrite the query as shown in the following example.
	`DECLARE @v int;`
	`SELECT @v = BusinessEntityID FROM`
	`(SELECT BusinessEntityID FROM HumanResources.Employee`
	`UNION ALL`
	`SELECT BusinessEntityID FROM HumanResources.EmployeeAddress) AS Test;`
	`SELECT @v;`	

The ODBC function {fn CONVERT()} uses the default date format of the language. For some languages, the default format is YDM, which can result in conversion errors when CONVERT() is combined with other functions, such as {fn CURDATE()}, that expect a YMD format.	The ODBC function {fn CONVERT()} uses style 121 (a language-independent YMD format) when converting to the ODBC data types SQL_TIMESTAMP, SQL_DATE, SQL_TIME, SQLDATE, SQL_TYPE_TIME, and SQL_TYPE_TIMESTAMP.	Low
The ODBC function {fn CURDATE()} returns only the date in the format	The ODBC function {fn CURDATE()} returns both date and time, for	Low

Compatibility-level setting of 90	Compatibility-level setting of 100	Possibility of impact
'YYYY-MM-DD'.	example 'YYYY-MM-DD hh:mm:ss.	
Datetime intrinsics such as DATEPART do not require string input values to be valid datetime literals. For example, SELECT DATEPART (year, '2007/05-30') compiles successfully.	Datetime intrinsics such as DATEPART require string input values to be valid datetime literals. Error 241 is returned when an invalid datetime literal is used.	Low

Differences Between Lower Compatibility Levels and Level 110

This section describes new behaviors introduced with compatibility level 110.

Compatibility-level setting of 100 or lower	Compatibility-level setting of 110
Common language runtime (CLR) database objects are executed with version 4 of the CLR. However, some behavior changes introduced in version 4 of the CLR are avoided. For more information, see What's New in CLR Integration.	CLR database objects are executed with version 4 of the CLR.
The XQuery functions **string-length** and **substring** count each surrogate as two characters.	The XQuery functions **string-length** and **substring** count each surrogate as one character.
PIVOT is allowed in a recursive common table expression (CTE) query. However, the query returns incorrect results when there are multiple rows per grouping.	PIVOT is not allowed in a recursive common table expression (CTE) query. An error is returned.
The RC4 algorithm is only supported for backward compatibility. New material can only be encrypted using RC4 or RC4_128 when the database is in compatibility level 90 or 100. (Not recommended.) In SQL Server 2012, material encrypted using RC4 or RC4_128 can be decrypted in any compatibility level.	New material cannot be encrypted using RC4 or RC4_128. Use a newer algorithm such as one of the AES algorithms instead. In SQL Server 2012, material encrypted using RC4 or RC4_128 can be decrypted in any compatibility level.
The default style for CAST and CONVERT operations on **time** and **datetime2** data types is 121 except when either type is used in a computed column expression. For	Under compatibility level 110, the default style for CAST and CONVERT operations on **time** and **datetime2** data types is always 121. If your query relies on the old

Compatibility-level setting of 100 or lower	Compatibility-level setting of 110

computed columns, the default style is 0. This behavior impacts computed columns when they are created, used in queries involving auto-parameterization, or used in constraint definitions.
The following example shows the difference between styles 0 and 121. It does not demonstrate the behavior described above. For more information about date and time styles, see CAST and CONVERT (Transact-SQL).
`CREATE TABLE t1 (c1 time(7), c2 datetime2);`
 	
`INSERT t1 (c1,c2) VALUES (GETDATE(), GETDATE());`
 	
`SELECT CONVERT(nvarchar(16),c1,0) AS TimeStyle0`
 	
`,CONVERT(nvarchar(16),c1,121)AS TimeStyle121`
`,CONVERT(nvarchar(32),c2,0) AS Datetime2Style0`
 	
`,CONVERT(nvarchar(32),c2,121)AS Datetime2Style121`
`FROM t1;`
 	
`-- Returns values such as the following.`
`TimeStyle0 TimeStyle121 Datetime2Style0 Datetime2Style121`
`--- --- - --- ---`
	behavior, use a compatibility level less than 110, or explicitly specify the 0 style in the affected query.
	Upgrading the database to compatibility level 110 will not change user data that has been stored to disk. You must manually correct this data as appropriate. For example, if you used SELECT INTO to create a table from a source that contained a computed column expression described above, the data (using style 0) would be stored rather than the computed column definition itself. You would need to manually update this data to match style 121.

Compatibility-level setting of 100 or lower	Compatibility-level setting of 110
---------- 3:15PM 15:15:35.8100000 Jun 7 2011 3:15PM 2011-06-07 15:15:35.8130000	
Any columns in remote tables of type **smalldatetime** that are referenced in a partitioned view are mapped as **datetime**. Corresponding columns in local tables (in the same ordinal position in the select list) must be of type **datetime**.	Any columns in remote tables of type **smalldatetime** that are referenced in a partitioned view are mapped as **smalldatetime**. Corresponding columns in local tables (in the same ordinal position in the select list) must be of type **smalldatetime**. After upgrading to 110, the distributed partitioned view will fail because of the data type mismatch. You can resolve this by changing the data type on the remote table to **datetime** or setting the compatibility level of the local database to 100 or lower.
SOUNDEX function implements the following rules. 1. If character_expression has any double letters, they are treated as one letter. 2. If a vowel (A, E, I, O, U) separates two consonants that have the same soundex code, the consonant to the right of the vowel is coded.	SOUNDEX function implements the following rules 1. If character_expression has any double letters, they are treated as one letter. 2. If character_expression has different letters side-by-side that have the same number in the soundex coding guide, they are treated as one letter. 3. If a vowel (A, E, I, O, U) separates two consonants that have the same soundex code, the consonant to the right of the vowel is coded. 4. If H or W separate two consonants that have the same soundex code, the consonant to the right of the vowel is not coded. The additional rules may cause the values computed by the SOUNDEX function to be different than the values computed under earlier compatibility levels. After upgrading to compatibility level 110, you may need to

Compatibility-level setting of 100 or lower	Compatibility-level setting of 110
	rebuild the indexes, heaps, or CHECK constraints that use the SOUNDEX function. For more information, see SOUNDEX (Transact-SQL)

Reserved Keywords

The compatibility setting also determines the keywords that are reserved by the Database Engine. The following table shows the reserved keywords that are introduced by each of the compatibility levels.

Compatibility-level setting	Reserved keywords
110	WITHIN GROUP, TRY_CONVERT, SEMANTICKEYPHRASETABLE, SEMANTICSIMILARITYDETAILSTABLE, SEMANTICSIMILARITYTABLE
100	CUBE, MERGE, ROLLUP
90	EXTERNAL, PIVOT, UNPIVOT, REVERT, TABLESAMPLE

At a given compatibility level, the reserved keywords include all of the keywords introduced at or below that level. Thus, for instance, for applications at level 110, all of the keywords listed in the preceding table are reserved. At the lower compatibility levels, level-100 keywords remain valid object names, but the level-110 language features corresponding to those keywords are unavailable.

Once introduced, a keyword remains reserved. For example, the reserved keyword PIVOT, which was introduced in compatibility level 90, is also reserved in levels 100 and 110.

If an application uses an identifier that is reserved as a keyword for its compatibility level, the application will fail. To work around this, enclose the identifier between either brackets (**[ ]**) or quotation marks (**" "**); for example, to upgrade an application that uses the identifier **EXTERNAL** to compatibility level 90, you could change the identifier to either **[EXTERNAL]** or **"EXTERNAL"**.

For more information, see Reserved Keywords (Transact-SQL).

Permissions

Requires ALTER permission on the database.

Examples

A. Changing the compatibility level

The following example changes the compatibility level of the database to `110`, SQL Server 2012.

```
ALTER DATABASE AdventureWorks2012
SET COMPATIBILITY_LEVEL = 110;
GO
```

See Also

ALTER DATABASE
Reserved Keywords
CREATE DATABASE
DATABASEPROPERTYEX
sys.databases (Transact-SQL)
sys.database_files

ALTER DATABASE AUDIT SPECIFICATION

Alters a database audit specification object using the SQL Server Audit feature. For more information, see Understanding SQL Server Audit.

 Transact-SQL Syntax Conventions

Syntax

```
ALTER DATABASE AUDIT SPECIFICATION audit_specification_name
{
   [ FOR SERVER AUDIT audit_name ]
   [ { { ADD | DROP } (
        { <audit_action_specification> | audit_action_group_name }
           )
    } [, ...n] ]
   [ WITH ( STATE = { ON | OFF } ) ]
}
[ ; ]
<audit_action_specification>::=
{
    <action_specification>[ ,...n ]ON [ class :: ] securable [ ( column [ ,...n ] ) ]
    BY principal [ ,...n ]
}
```

<action_specification>::=

```
{
    action [ ( column [ ,...n ] ) ]
}
```

Arguments

audit_specification_name

The name of the audit specification.

audit_name

The name of the audit to which this specification is applied.

audit_action_specification

Name of one or more database-level auditable actions. For a list of audit action groups, see SQL Server Audit Action Groups and Actions.

audit_action_group_name

Name of one or more groups of database-level auditable actions. For a list of audit action groups, see SQL Server Audit Action Groups and Actions.

class

Class name (if applicable) on the securable.

securable

Table, view, or other securable object in the database on which to apply the audit action or audit action group. For more information, see Securables.

column

Column name (if applicable) on the securable.

principal

Name of SQL Server principal on which to apply the audit action or audit action group. For more information, see Principals (Database Engine).

WITH (STATE = { ON | OFF })

Enables or disables the audit from collecting records for this audit specification. Audit specification state changes must be done outside a user transaction and may not have other changes in the same statement when the transition is ON to OFF.

Remarks

Database audit specifications are non-securable objects that reside in a given database. You must set the state of an audit specification to the OFF option in order to make changes to a database audit specification. If ALTER DATABASE AUDIT SPECIFICATION is executed when an audit is enabled with any options other than STATE=OFF, you will receive an error message. For more information, see tempdb Database.

Permissions

Users with the ALTER ANY DATABASE AUDIT permission can alter database audit specifications and bind them to any audit.

After a database audit specification is created, it can be viewed by principals with the CONTROL SERVER, or ALTER ANY DATABASE AUDIT permissions, the sysadmin account, or principals having explicit access to the audit.

Examples

The following example alters a database audit specification called `HIPPA_Audit_DB_Specification` that audits the `SELECT` statements by the `dbo` user, for a SQL Server audit called `HIPPA_Audit`.

```
ALTER DATABASE AUDIT SPECIFICATION HIPPA_Audit_DB_Specification
FOR SERVER AUDIT HIPPA_Audit
    ADD (SELECT
         ON Table1(Column1)
         BY dbo)
    WITH STATE = ON;
GO
```

For a full example about how to create an audit, see Understanding SQL Server Audit.

Updated content
Corrected the Permissions section.

See Also

CREATE SERVER AUDIT (Transact-SQL)
ALTER SERVER AUDIT (Transact-SQL)
DROP SERVER AUDIT (Transact-SQL)
CREATE SERVER AUDIT SPECIFICATION (Transact-SQL)
ALTER SERVER AUDIT SPECIFICATION (Transact-SQL)
DROP SERVER AUDIT SPECIFICATION (Transact-SQL)
CREATE DATABASE AUDIT SPECIFICATION (Transact-SQL)
DROP DATABASE AUDIT SPECIFICATION (Transact-SQL)
ALTER AUTHORIZATION (Transact-SQL)
fn_get_audit_file (Transact-SQL)
sys.server_audits (Transact-SQL)

sys.server_file_audits (Transact-SQL)
sys.server_audit_specifications (Transact-SQL)
sys.server_audit_specifications_details (Transact-SQL)
sys.database_audit_specifications (Transact-SQL)
sys.audit_database_specification_details (Transact-SQL)
sys.dm_server_audit_status
sys.dm_audit_actions
Create a Server Audit and Server Audit Specification

ALTER DATABASE ENCRYPTION KEY

Alters an encryption key and certificate that is used for transparently encrypting a database. For more information about transparent database encryption, see Understanding Transparent Data Encryption (TDE).

 Transact-SQL Syntax Conventions

Syntax

```
ALTER DATABASE ENCRYPTION KEY
    REGENERATE WITH ALGORITHM = { AES_128 | AES_192 | AES_256 | TRIPLE_DES_3KEY }
  |
  ENCRYPTION BY SERVER
  {
     CERTIFICATE Encryptor_Name |
     ASYMMETRIC KEY Encryptor_Name
  }
[ ; ]
```

Arguments

REGENERATE WITH ALGORITHM = { AES_128 | AES_192 | AES_256 | TRIPLE_DES_3KEY }

Specifies the encryption algorithm that is used for the encryption key.

ENCRYPTION BY SERVER CERTIFICATE Encryptor_Name

Specifies the name of the certificate used to encrypt the database encryption key.

ENCRYPTION BY SERVER ASYMMETRIC KEY Encryptor_Name

Specifies the name of the asymmetric key used to encrypt the database encryption key.

Remarks

The certificate or asymmetric key that is used to encrypt the database encryption key must be located in the master system database.

The database encryption key does not have to be regenerated when a database owner (dbo) is changed.

After a database encryption key has been modified twice, a log backup must be performed before the database encryption key can be modified again.

Permissions

Requires CONTROL permission on the database and VIEW DEFINITION permission on the certificate or asymmetric key that is used to encrypt the database encryption key.

Examples

The following example alters the database encryption key to use the `AES_256` algorithm.

```
USE AdventureWorks2012;
GO
ALTER DATABASE ENCRYPTION KEY
REGENERATE WITH ALGORITHM = AES_256;
GO
```

See Also

Understanding Transparent Data Encryption (TDE)
SQL Server Encryption
SQL Server and Database Encryption Keys (Database Engine)
Encryption Hierarchy
ALTER DATABASE SET Options (Transact-SQL)
CREATE DATABASE ENCRYPTION KEY (Transact-SQL)
DROP DATABASE ENCRYPTION KEY (Transact-SQL)
sys.dm_database_encryption_keys

ALTER ENDPOINT

Enables modifying an existing endpoint in the following ways:

- By adding a new method to an existing endpoint.
- By modifying or dropping an existing method from the endpoint.
- By changing the properties of an endpoint.

Note

This topic describes the syntax and arguments that are specific to ALTER ENDPOINT. For descriptions of the arguments that are common to both CREATE ENDPOINT and ALTER ENDPOINT, see CREATE ENDPOINT (Transact-SQL).

Native XML Web Services (SOAP/HTTP endpoints) is removed beginning in SQL Server 2012.

 Transact-SQL Syntax Conventions

Syntax

```
ALTER ENDPOINT endPointName [ AUTHORIZATION login ]
[ STATE = { STARTED | STOPPED | DISABLED } ]
[ AS { TCP } ( <protocol_specific_items> ) ]
[ FOR { TSQL | SERVICE_BROKER | DATABASE_MIRRORING } (
   <language_specific_items>
      ) ]

<AS TCP_protocol_specific_arguments> ::=
AS TCP (
  LISTENER_PORT = listenerPort
  [ [ , ] LISTENER_IP = ALL | ( 4-part-ip ) | ( "ip_address_v6" ) ]
)
<FOR SERVICE_BROKER_language_specific_arguments> ::=
FOR SERVICE_BROKER (
   [ AUTHENTICATION = {
      WINDOWS [ { NTLM | KERBEROS | NEGOTIATE } ]
      | CERTIFICATE certificate_name
      | WINDOWS [ { NTLM | KERBEROS | NEGOTIATE } ] CERTIFICATE certificate_name
      | CERTIFICATE certificate_name WINDOWS [ { NTLM | KERBEROS | NEGOTIATE } ]
    } ]
   [ , ENCRYPTION = { DISABLED
       |
        {{SUPPORTED | REQUIRED }
       [ ALGORITHM { RC4 | AES | AES RC4 | RC4 AES } ] }
   ]

  [ , MESSAGE_FORWARDING = {ENABLED | DISABLED} ]
  [ , MESSAGE_FORWARD_SIZE = forwardSize
```

```
)

<FOR DATABASE_MIRRORING_language_specific_arguments> ::=
FOR DATABASE_MIRRORING (
  [ AUTHENTICATION = {
    WINDOWS [ { NTLM | KERBEROS | NEGOTIATE } ]
    | CERTIFICATE certificate_name
    | WINDOWS [ { NTLM | KERBEROS | NEGOTIATE } ] CERTIFICATE certificate_name
    | CERTIFICATE certificate_name WINDOWS [ { NTLM | KERBEROS | NEGOTIATE } ]
   } ]
  [ , ENCRYPTION = { DISABLED
     |
      {{SUPPORTED | REQUIRED }
     [ ALGORITHM { RC4 | AES | AES RC4 | RC4 AES } ] }
   ]
  [ , ] ROLE = { WITNESS | PARTNER | ALL }
)
```

Arguments

Note
The following arguments are specific to ALTER ENDPOINT. For descriptions of the remaining arguments, see CREATE ENDPOINT (Transact-SQL).

AS { TCP }

You cannot change the transport protocol with **ALTER ENDPOINT**.

AUTHORIZATION login

The **AUTHORIZATION** option is not available in **ALTER ENDPOINT**. Ownership can only be assigned when the endpoint is created.

FOR { TSQL | SERVICE_BROKER | DATABASE_MIRRORING }

You cannot change the payload type with **ALTER ENDPOINT**.

Remarks

When you use ALTER ENDPOINT, specify only those parameters that you want to update. All properties of an existing endpoint remain the same unless you explicitly change them.

The ENDPOINT DDL statements cannot be executed inside a user transaction.

For information on choosing an encryption algorithm for use with an endpoint, see Choosing an Encryption Algorithm.

Note

- The RC4 algorithm is only supported for backward compatibility. New material can only be encrypted using RC4 or RC4_128 when the database is in compatibility level 90 or 100. (Not recommended.) Use a newer algorithm such as one of the AES algorithms instead. In SQL Server 2012 material encrypted using RC4 or RC4_128 can be decrypted in any compatibility level.
- RC4 is a relatively weak algorithm, and AES is a relatively strong algorithm. But AES is considerably slower than RC4. If security is a higher priority for you than speed, we recommend you use AES.

Permissions

User must be a member of the **sysadmin** fixed server role, the owner of the endpoint, or have been granted ALTER ANY ENDPOINT permission.

To change ownership of an existing endpoint, you must use the ALTER AUTHORIZATION statement. For more information, see ALTER AUTHORIZATION (Transact-SQL).

For more information, see GRANT Endpoint Permissions (Transact-SQL).

See Also

DROP ENDPOINT (Transact-SQL)

eventdata (Transact-SQL)

ALTER EVENT SESSION

Starts or stops an event session or changes an event session configuration.

Transact-SQL Syntax Conventions

Syntax

```
ALTER EVENT SESSION event_session_name
ON SERVER
{
   [ [ {  <add_drop_event> [ ,...n] }
      | { <add_drop_event_target> [ ,...n ] } ]
   [ WITH ( <event_session_options> [ ,...n ] ) ]
   ]
   | [ STATE = { START | STOP } ]
}

<add_drop_event>::=
```

```
{
   [ ADD EVENT <event_specifier>
      [ ( {
            [ SET { event_customizable_attribute = <value> [ ,...n ] } ]
            [ ACTION ( { [event_module_guid].event_package_name.action_name [ ,...n ] } )
]
            [ WHERE <predicate_expression> ]
      } ) ]
   ]
   | DROP EVENT <event_specifier> }

<event_specifier> ::=
{
[event_module_guid].event_package_name.event_name
}

<predicate_expression> ::=
{
   [ NOT ] <predicate_factor> | { ( <predicate_expression> ) }
   [ { AND | OR } [ NOT ] { <predicate_factor> | ( <predicate_expression> ) } ]
   [ ,...n ]
}

<predicate_factor>::=
{
   <predicate_leaf> | ( <predicate_expression> )
}

<predicate_leaf>::=
{
     <predicate_source_declaration> { = | < > | ! = | > | > = | < | < = } <value>
   | [event_module_guid].event_package_name.predicate_compare_name (
<predicate_source_declaration>, <value> )
}

<predicate_source_declaration>::=
```

```
{
   event_field_name | (
[event_module_guid].event_package_name.predicate_source_name )
}

<value>::=
{
   number | 'string'
}

<add_drop_event_target>::=
{
   ADD TARGET <event_target_specifier>
      [ ( SET { target_parameter_name = <value> [ ,...n] } ) ]
   | DROP TARGET <event_target_specifier>
}

<event_target_specifier>::=
{
   [event_module_guid].event_package_name.target_name
}

<event_session_options>::=
{
   [    MAX_MEMORY = size [ KB | MB ] ]
   [ [,] EVENT_RETENTION_MODE = { ALLOW_SINGLE_EVENT_LOSS |
ALLOW_MULTIPLE_EVENT_LOSS | NO_EVENT_LOSS } ]
   [ [,] MAX_DISPATCH_LATENCY = { seconds SECONDS | INFINITE } ]
   [ [,] MAX_EVENT_SIZE = size [ KB | MB ] ]
   [ [,] MEMORY_PARTITION_MODE = { NONE | PER_NODE | PER_CPU } ]
   [ [,] TRACK_CAUSALITY = { ON | OFF } ]
   [ [,] STARTUP_STATE = { ON | OFF } ]
}
```

Arguments

Term	Definition
event_session_name	Is the name of an existing event session.
STATE = START | STOP	Starts or stops the event session. This argument is only valid when ALTER EVENT SESSION is applied to an event session object.
ADD EVENT <event_specifier>	Associates the event identified by <event_specifier> with the event session.
[event_module_guid].event_package_name.event_name	Is the name of an event in an event package, where: • event_module_guid is the GUID for the module that contains the event. • event_package_name is the package that contains the action object. • event_name is the event object. Events appear in the sys.dm_xe_objects view as object_type 'event'.
SET { event_customizable_attribute = <value> [,...n] }	Specifies customizable attributes for the event. Customizable attributes appear in the sys.dm_xe_object_columns view as column_type 'customizable ' and object_name = event_name.
ACTION ({ [event_module_guid].event_package_name.action_name [,...n] })	Is the action to associate with the event session, where: • event_module_guid is the GUID for the module that contains the event. • event_package_name is the package that contains the action object. • action_name is the action object. Actions appear in the

	sys.dm_xe_objects view as object_type 'action'.
WHERE <predicate_expression>	Specifies the predicate expression used to determine if an event should be processed. If <predicate_expression> is true, the event is processed further by the actions and targets for the session. If <predicate_expression> is false, the event is dropped by the session before being processed by the actions and targets for the session. Predicate expressions are limited to 3000 characters, which limits string arguments.
event_field_name	Is the name of the event field that identifies the predicate source.
[event_module_guid].event_package_name.predicate_source_name	Is the name of the global predicate source where: • event_module_guid is the GUID for the module that contains the event. • event_package_name is the package that contains the predicate object. • predicate_source_name is defined in the sys.dm_xe_objects view as object_type 'pred_source'.
[event_module_guid].event_package_name.predicate_compare_name	Is the name of the predicate object to associate with the event, where: • event_module_guid is the GUID for the module that contains the event. • event_package_name is the package that contains the predicate object. • predicate_compare_name is a global source defined in the

	sys.dm_xe_objects view as object_type 'pred_compare'.
DROP EVENT <event_specifier>	Drops the event identified by <event_specifier>. <event_specifier> must be valid in the event session.
ADD TARGET <event_target_specifier>	Associates the target identified by <event_target_specifier> with the event session.
[event_module_guid].event_package_name.target_name	Is the name of a target in the event session, where: • event_module_guid is the GUID for the module that contains the event. • event_package_name is the package that contains the action object. • target_name is the action. Actions appear in sys.dm_xe_objects view as object_type 'target'.
SET { target_parameter_name = <value> [, ...n] }	Sets a target parameter. Target parameters appear in the sys.dm_xe_object_columns view as column_type 'customizable' and object_name = target_name.  **Important** If you are using the ring buffer target, we recommend that you set the max_memory target parameter to 2048 kilobytes (KB) to help avoid possible data truncation of the XML output. For more information about when to use the different target types, see SQL Server Extended Events Targets.

DROP TARGET <event_target_specifier>	Drops the target identified by <event_target_specifier>. <event_target_specifier> must be valid in the event session.
EVENT_RETENTION_MODE = { ALLOW_SINGLE_EVENT_LOSS | ALLOW_MULTIPLE_EVENT_LOSS | NO_EVENT_LOSS }	Specifies the event retention mode to use for handling event loss. **ALLOW_SINGLE_EVENT_LOSS** An event can be lost from the session. A single event is only dropped when all the event buffers are full. Losing a single event when event buffers are full allows for acceptable SQL Server performance characteristics, while minimizing the loss of data in the processed event stream. **ALLOW_MULTIPLE_EVENT_LOSS** Full event buffers containing multiple events can be lost from the session. The number of events lost is dependent upon the memory size allocated to the session, the partitioning of the memory, and the size of the events in the buffer. This option minimizes performance impact on the server when event buffers are quickly filled, but large numbers of events can

	be lost from the session. **NO_EVENT_LOSS** No event loss is allowed. This option ensures that all events raised will be retained. Using this option forces all tasks that fire events to wait until space is available in an event buffer. This may cause detectable performance issues while the event session is active. User connections may stall while waiting for events to be flushed from the buffer.
MAX_DISPATCH_LATENCY = { seconds SECONDS | INFINITE }	Specifies the amount of time that events are buffered in memory before being dispatched to event session targets. The minimum latency value is 1 second. However, 0 can be used to specify INFINITE latency. By default, this value is set to 30 seconds. **seconds SECONDS** The time, in seconds, to wait before starting to flush buffers to targets. seconds is a whole number. **INFINITE** Flush buffers to targets only when the buffers are full, or when the event session closes. **Note** MAX_DISPATCH_LATENCY = 0

<table>
<tr><td></td><td>SECONDS is equivalent to MAX_DISPATCH_LATENCY = INFINITE.</td></tr>
<tr><td>MAX_EVENT_SIZE = size [KB | MB]</td><td>Specifies the maximum allowable size for events. MAX_EVENT_SIZE should only be set to allow single events larger than MAX_MEMORY; setting it to less than MAX_MEMORY will raise an error. size is a whole number and can be a kilobyte (KB) or a megabyte (MB) value. If size is specified in kilobytes, the minimum allowable size is 64 KB. When MAX_EVENT_SIZE is set, two buffers of size are created in addition to MAX_MEMORY. This means that the total memory used for event buffering is MAX_MEMORY + 2 * MAX_EVENT_SIZE.</td></tr>
<tr><td>MEMORY_PARTITION_MODE = { NONE | PER_NODE | PER_CPU }</td><td>Specifies the location where event buffers are created.
NONE
A single set of buffers is created within the SQL Server instance.
<table><tr><td>PER_NODE</td><td>A set of buffers is created for each NUMA node.</td></tr><tr><td>PER_CPU</td><td>A set of buffers is created for each CPU.</td></tr></table></td></tr>
</table>

<table>
<tr><td></td><td></td></tr>
<tr><td>TRACK_CAUSALITY = { ON | OFF }</td><td>Specifies whether or not causality is tracked. If enabled, causality allows related events on different server connections to be correlated together.</td></tr>
<tr><td>STARTUP_STATE = { ON | OFF }</td><td>Specifies whether or not to start this event session automatically when SQL Server starts.

nNote
If STARTUP_STATE = ON, the event session will only start if SQL Server is stopped and then restarted.

<table><tr><td>Term</td><td>Definition</td></tr><tr><td>ON</td><td>The event session is started at startup.</td></tr><tr><td>OFF</td><td>The event session is not started at startup.</td></tr></table></td></tr>
</table>

Remarks

The ADD and DROP arguments cannot be used in the same statement.

Permissions

Requires the ALTER ANY EVENT SESSION permission.

Examples

The following example starts an event session, obtains some live session statistics, and then adds two events to the existing session.

```
-- Start the event session
ALTER EVENT SESSION test_session
ON SERVER
```

```
STATE = start
GO
-- Obtain live session statistics
SELECT * FROM sys.dm_xe_sessions
SELECT * FROM sys.dm_xe_session_events
GO

-- Add new events to the session
ALTER EVENT SESSION test_session ON SERVER
ADD EVENT sqlserver.database_transaction_begin,
ADD EVENT sqlserver.database_transaction_end
GO
```

See Also

CREATE EVENT SESSION (Transact-SQL)
DROP EVENT SESSION (Transact-SQL)
Extended Event Targets
sys.server_event_sessions
sys.dm_xe_objects
sys.dm_xe_object_columns

ALTER FULLTEXT CATALOG

Changes the properties of a full-text catalog.

 Transact-SQL Syntax Conventions

Syntax

```
ALTER FULLTEXT CATALOG catalog_name
{ REBUILD [ WITH ACCENT_SENSITIVITY = { ON | OFF } ]
| REORGANIZE
| AS DEFAULT
}
```

Arguments

catalog_name

Specifies the name of the catalog to be modified. If a catalog with the specified name does

not exist, Microsoft SQL Server returns an error and does not perform the ALTER operation.

REBUILD

Tells SQL Server to rebuild the entire catalog. When a catalog is rebuilt, the existing catalog is deleted and a new catalog is created in its place. All the tables that have full-text indexing references are associated with the new catalog. Rebuilding resets the full-text metadata in the database system tables.

WITH ACCENT_SENSITIVITY = {ON|OFF}

Specifies if the catalog to be altered is accent-sensitive or accent-insensitive for full-text indexing and querying.

To determine the current accent-sensitivity property setting of a full-text catalog, use the FULLTEXTCATALOGPROPERTY function with the **accentsensitivity** property value against catalog_name. If the function returns '1', the full-text catalog is accent sensitive; if the function returns '0', the catalog is not accent sensitive.

The catalog and database default accent sensitivity are the same.

REORGANIZE

Tells SQL Server to perform a *master merge*, which involves merging the smaller indexes created in the process of indexing into one large index. Merging the full-text index fragments can improve performance and free up disk and memory resources. If there are frequent changes to the full-text catalog, use this command periodically to reorganize the full-text catalog.

REORGANIZE also optimizes internal index and catalog structures.

Keep in mind that, depending on the amount of indexed data, a master merge may take some time to complete. Master merging a large amount of data can create a long running transaction, delaying truncation of the transaction log during checkpoint. In this case, the transaction log might grow significantly under the full recovery model. As a best practice, ensure that your transaction log contains sufficient space for a long-running transaction before reorganizing a large full-text index in a database that uses the full recovery model. For more information, see Managing the Size of the Transaction Log File.

AS DEFAULT

Specifies that this catalog is the default catalog. When full-text indexes are created with no specified catalogs, the default catalog is used. If there is an existing default full-text catalog, setting this catalog AS DEFAULT will override the existing default.

Permissions

User must have ALTER permission on the full-text catalog, or be a member of the **db_owner**, **db_ddladmin** fixed database roles, or sysadmin fixed server role.

Note

To use ALTER FULLTEXT CATALOG AS DEFAULT, the user must have ALTER permission on the full-text catalog and CREATE FULLTEXT CATALOG permission on the database.

Examples

The following example changes the `accentsensitivity` property of the default full-text catalog `ftCatalog`, which is accent sensitive.

```
--Change to accent insensitive
USE AdventureWorks;
GO
ALTER FULLTEXT CATALOG ftCatalog
REBUILD WITH ACCENT_SENSITIVITY=OFF;
GO
-- Check Accentsensitivity
SELECT FULLTEXTCATALOGPROPERTY('ftCatalog', 'accentsensitivity');
GO
--Returned 0, which means the catalog is not accent sensitive.
```

See Also

sys.fulltext_catalogs (Transact-SQL)
Full-Text Search
DROP FULLTEXT CATALOG
Full-Text Search

ALTER FULLTEXT INDEX

Changes the properties of a full-text index.

 Transact-SQL Syntax Conventions

Syntax

```
ALTER FULLTEXT INDEX ON table_name
  { ENABLE
  | DISABLE
  | SET CHANGE_TRACKING [ = ] { MANUAL | AUTO | OFF }
  | ADD ( column_name
    [ TYPE COLUMN type_column_name ]
    [ LANGUAGE language_term ]
    [ STATISTICAL_SEMANTICS ]
```

[,...n])
  [WITH NO POPULATION]
 | ALTER COLUMN **column_name**
  { ADD | DROP } STATISTICAL_SEMANTICS
  [WITH NO POPULATION]
 | DROP (**column_name** [,...n])
  [WITH NO POPULATION]
 | START { FULL | INCREMENTAL | UPDATE } POPULATION
 | {STOP | PAUSE | RESUME } POPULATION
 | SET STOPLIST [=] { OFF| SYSTEM | **stoplist_name** }
  [WITH NO POPULATION]
 | SET SEARCH PROPERTY LIST [=] { OFF | **property_list_name** }
  [WITH NO POPULATION]
 }
[;]

Arguments

table_name

Is the name of the table or indexed view that contains the column or columns included in the full-text index. Specifying database and table owner names is optional.

ENABLE | DISABLE

Tells SQL Server whether to gather full-text index data for table_name. ENABLE activates the full-text index; DISABLE turns off the full-text index. The table will not support full-text queries while the index is disabled.

Disabling a full-text index allows you to turn off change tracking but keep the full-text index, which you can reactivate at any time using ENABLE. When the full-text index is disabled, the full-text index metadata remains in the system tables. If CHANGE_TRACKING is in the enabled state (automatic or manual update) when the full-text index is disabled, the state of the index freezes, any ongoing crawl stops, and new changes to the table data are not tracked or propagated to the index.

SET CHANGE_TRACKING {MANUAL | AUTO | OFF}

Specifies whether changes (updates, deletes, or inserts) made to table columns that are covered by the full-text index will be propagated by SQL Server to the full-text index. Data changes through WRITETEXT and UPDATETEXT are not reflected in the full-text index, and are not picked up with change tracking.

Note

For information about the interaction of change tracking and WITH NO POPULATION, see "Remarks,"

later in this topic.

MANUAL

Specifies that the tracked changes will be propagated manually by calling the ALTER FULLTEXT INDEX ... START UPDATE POPULATION Transact-SQL statement (*manual population*). You can use SQL Server Agent to call this Transact-SQL statement periodically.

AUTO

Specifies that the tracked changes will be propagated automatically as data is modified in the base table (*automatic population*). Although changes are propagated automatically, these changes might not be reflected immediately in the full-text index. AUTO is the default.

OFF

Specifies that SQL Server will not keep a list of changes to the indexed data.

ADD | DROP column_name

Specifies the columns to be added or deleted from a full-text index. The column or columns must be of type **char**, **varchar**, **nchar**, **nvarchar**, **text**, **ntext**, **image**, **xml**, **varbinary**, or **varbinary(max)**.

Use the DROP clause only on columns that have been enabled previously for full-text indexing.

Use TYPE COLUMN and LANGUAGE with the ADD clause to set these properties on the column_name. When a column is added, the full-text index on the table must be repopulated in order for full-text queries against this column to work.

Note

Whether the full-text index is populated after a column is added or dropped from a full-text index depends on whether change-tracking is enabled and whether WITH NO POPULATION is specified. For more information, see "Remarks," later in this topic.

TYPE COLUMN type_column_name

Specifies the name of a table column, type_column_name, that is used to hold the document type for a **varbinary**, **varbinary(max)**, or **image** document. This column, known as the type column, contains a user-supplied file extension (.doc, .pdf, .xls, and so forth). The type column must be of type **char**, **nchar**, **varchar**, or **nvarchar**.

Specify TYPE COLUMN type_column_name only if column_name specifies a **varbinary**, **varbinary(max)** or **image** column, in which data is stored as binary data; otherwise, SQL Server returns an error.

Note

At indexing time, the Full-Text Engine uses the abbreviation in the type column of each table row to identify which full-text search filter to use for the document in column_name. The filter loads the document as a binary stream, removes the formatting information, and sends the text from the document to the word-breaker component. For more information, see Full-Text Search Filters.

LANGUAGE language_term

Is the language of the data stored in **column_name**.

language_term is optional and can be specified as a string, integer, or hexadecimal value corresponding to the locale identifier (LCID) of a language. If language_term is specified, the language it represents will be applied to all elements of the search condition. If no value is specified, the default full-text language of the SQL Server instance is used.

Use the **sp_configure** stored procedure to access information about the default full-text language of the SQL Server instance.

When specified as a string, language_term corresponds to the **alias** column value in the **syslanguages** system table. The string must be enclosed in single quotation marks, as in 'language_term'. When specified as an integer, language_term is the actual LCID that identifies the language. When specified as a hexadecimal value, language_term is 0x followed by the hex value of the LCID. The hex value must not exceed eight digits, including leading zeros.

If the value is in double-byte character set (DBCS) format, SQL Server will convert it to Unicode.

Resources, such as word breakers and stemmers, must be enabled for the language specified as language_term. If such resources do not support the specified language, SQL Server returns an error.

For non-BLOB and non-XML columns containing text data in multiple languages, or for cases when the language of the text stored in the column is unknown, use the neutral (0x0) language resource. For documents stored in XML- or BLOB-type columns, the language encoding within the document will be used at indexing time. For example, in XML columns, the xml:lang attribute in XML documents will identify the language. At query time, the value previously specified in language_term becomes the default language used for full-text queries unless language_term is specified as part of a full-text query.

STATISTICAL_SEMANTICS

Creates the additional key phrase and document similarity indexes that are part of statistical semantic indexing. For more information, see Semantic Search.

[,...n]

Indicates that multiple columns may be specified for the ADD, ALTER, or DROP clauses. When multiple columns are specified, separate these columns with commas.

WITH NO POPULATION

Specifies that the full-text index will not be populated after an ADD or DROP column operation or a SET STOPLIST operation. The index will only be populated if the user executes a START...POPULATION command.

When NO POPULATION is specified, SQL Server does not populate an index. The index is populated only after the user gives an ALTER FULLTEXT INDEX...START POPULATION command. When NO POPULATION is not specified, SQL Server populates the index.

If CHANGE_TRACKING is enabled and WITH NO POPULATION is specified, SQL Server returns an error. If CHANGE_TRACKING is enabled and WITH NO POPULATION is not specified, SQL Server performs a full population on the index.

Note

For more information about the interaction of change tracking and WITH NO POPULATION, see "Remarks," later in this topic.

{ADD | DROP } STATISTICAL_SEMANTICS

Enables or disables statistical semantic indexing for the specified columns. For more information, see Semantic Search.

START {FULL|INCREMENTAL|UPDATE} POPULATION

Tells SQL Server to begin population of the full-text index of table_name. If a full-text index population is already in progress, SQL Server returns a warning and does not start a new population.

FULL

Specifies that every row of the table be retrieved for full-text indexing even if the rows have already been indexed.

INCREMENTAL

Specifies that only the modified rows since the last population be retrieved for full-text indexing. INCREMENTAL can be applied only if the table has a column of the type **timestamp**. If a table in the full-text catalog does not contain a column of the type **timestamp**, the table undergoes a FULL population.

UPDATE

Specifies the processing of all insertions, updates, or deletions since the last time the change-tracking index was updated. Change-tracking population must be enabled on a table, but the background update index or the auto change tracking should not be turned on.

{STOP | PAUSE | RESUME } POPULATION

Stops, or pauses any population in progress; or stops or resumes any paused population.

STOP POPULATION does not stop auto change tracking or background update index. To stop change tracking, use SET CHANGE_TRACKING OFF.

PAUSE POPULATION and RESUME POPULATION can only be used for full populations. They are not relevant to other population types because the other populations resume crawls from where the crawl stopped.

SET STOPLIST { OFF| SYSTEM | stoplist_name }

Changes the full-text stoplist that is associated with the index, if any.

OFF

Specifies that no stoplist be associated with the full-text index.

SYSTEM

Specifies that the default full-text system STOPLIST should be used for this full-text index.

stoplist_name

Specifies the name of the stoplist to be associated with the full-text index.

For more information, see Stopwords and Stoplists.

SET SEARCH PROPERTY LIST { OFF | property_list_name } [WITH NO POPULATION]

Changes the search property list that is associated with the index, if any.

OFF

Specifies that no property list be associated with the full-text index. When you turn off the search property list of a full-text index (ALTER FULLTEXT INDEX … SET SEARCH PROPERTY LIST OFF), property searching on the base table is no longer possible.

By default, when you turn off an existing search property list, the full-text index automatically repopulates. If you specify WITH NO POPULATION when you turn off the search property list, automatic repopulation does not occur. However, we recommend that you eventually run a full population on this full-text index at your convenience. Repopulating the full-text index removes the property-specific metadata of each dropped search property, making the full-text index smaller and more efficient.

property_list_name

Specifies the name of the search property list to be associated with the full-text index.

Adding a search property list to a full-text index requires repopulating the index to index the search properties that are registered for the associated search property list. If you specify WITH NO POPULATION when adding the search property list, you will need to run a population on the index, at an appropriate time.

Important

If the full-text index was previously associated with a different search it must be rebuilt property list in order to bring the index into a consistent state. The index is truncated immediately and is empty until the full population runs. For more information about when changing the search property list causes rebuilding, see "Remarks," later in this topic.

Note

You can associate a given search property list with more than one full-text index in the same database.

To find the search property lists on the current database

- sys.registered_search_property_lists

For more information about search property lists, see Using Property Lists to Search

for Document Properties.

Remarks

Interactions of Change Tracking and NO POPULATION Parameter

Whether the full-text index is populated depends on whether change-tracking is enabled and whether WITH NO POPULATION is specified in the ALTER FULLTEXT INDEX statement. The following table summarizes the result of their interaction.

Change Tracking	WITH NO POPULATION	Result
Not Enabled	Not specified	A full population is performed on the index.
Not Enabled	Specified	No population of the index occurs until an ALTER FULLTEXT INDEX...START POPULATION statement is issued.
Enabled	Specified	An error is raised, and the index is not altered.
Enabled	Not specified	A full population is performed on the index.

For more information about populating full-text indexes, see Full-Text Index Population.

Changing the Search Property List Causes Rebuilding the Index

The first time that a full-text index is associated with a search property list, the index must be repopulated to index property-specific search terms. The existing index data is not truncated.

However, if you associate the full-text index with a different property list, the index is rebuilt. Rebuilding immediately truncates the full-text index, removing all existing data, and the index must be repopulated. While the population progresses, full-text queries on the base table search only on the the table rows that have already been indexed by the population. The repopulated index data will include metadata from the registered properties of the newly added search property list.

Scenarios that cause rebuilding include:

- Switching directly to a different search property list (see "Scenario A," later in this section).
- Turning off the search property list and later associating the index with any search property list (see "Scenario B," later in this section)

Note

For more information about how full-text search works with search property lists, see Using Search Property Lists to Search for Properties (Full-Text Search). For information about full populations, see Full-Text Index Population.

Scenario A: Switching Directly to a Different Search Property List

1. A full-text index is created on `table_1` with a search property list `spl_1`:

```
CREATE FULLTEXT INDEX ON table_1 (column_name) KEY INDEX
unique_key_index
   WITH SEARCH PROPERTY LIST=spl_1,
   CHANGE_TRACKING OFF, NO POPULATION;
```

2. A full population is run on the full-text index:

```
ALTER FULLTEXT INDEX ON table_1 START FULL POPULATION;
```

3. The full-text index is later associated a different search property list, `spl_2`, using the following statement:

```
ALTER FULLTEXT INDEX ON table_1 SET SEARCH PROPERTY LIST spl_2;
```

   This statement causes a full population, the default behavior. However, before beginning this population, the Full-Text Engine automatically truncates the index.

Scenario B: Turning Off the Search Property List and Later Associating the Index with Any Search Property List

1. A full-text index is created on `table_1` with a search property list `spl_1`, followed by an automatic full population (the default behavior):

```
CREATE FULLTEXT INDEX ON table_1 (column_name) KEY INDEX
unique_key_index
   WITH SEARCH PROPERTY LIST=spl_1;
```

2. The search property list is turned off, as follows:

```
ALTER FULLTEXT INDEX ON table_1
   SET SEARCH PROPERTY LIST OFF WITH NO POPULATION;
```

3. The full-text index is once more associated either the same search property list or a different one.

   For example the following statement re-associates the full-text index with the original search property list, `spl_1`:

```
ALTER FULLTEXT INDEX ON table_1 SET SEARCH PROPERTY LIST spl_1;
```

   This statement starts a full population, the default behavior.

> **Note**
> The rebuild would also be required for a different search property list, such as `spl_2`.

Permissions

The user must have ALTER permission on the table or indexed view, or be a member of the **sysadmin** fixed server role, or the **db_ddladmin** or **db_owner** fixed database roles.

If SET STOPLIST is specified, the user must have REFERENCES permission on the stoplist. If SET SEARCH PROPERTY LIST is specified, the user must have REFERENCES permission on the search property list. The owner of the specified stoplist or search property list can grant REFERENCES permission, if the owner has ALTER FULLTEXT CATALOG permissions.

Note
The public is granted REFERENCES permission to the default stoplist that is shipped with SQL Server.

Examples

A. Setting manual change tracking

The following example sets manual change tracking on the full-text index on the `JobCandidate` table of the `AdventureWorks` database.

```
USE AdventureWorks;
GO
ALTER FULLTEXT INDEX ON HumanResources.JobCandidate
   SET CHANGE_TRACKING MANUAL;
GO
```

B. Associating a property list with a full-text index

The following example associates the `DocumentPropertyList` property list with the full-text index on the `Production.Document` table of the `AdventureWorks` database. This ALTER FULLTEXT INDEX statement starts a full population, which is the default behavior of the SET SEARCH PROPERTY LIST clause.

Note
For an example that creates the `DocumentPropertyList` property list, see CREATE SEARCH PROPERTY LIST (Transact-SQL).

```
USE AdventureWorks;
GO
ALTER FULLTEXT INDEX ON Production.Document
   SET SEARCH PROPERTY LIST DocumentPropertyList;
GO
```

C. Removing a search property list

The following example removes the `DocumentPropertyList` property list from the full-text index on the `Production.Document` table of the `AdventureWorks` database. In this example, there is no hurry for removing the properties from the index, so the WITH NO POPULATION

option is specified. However, property-level searching is longer allowed against this full-text index.

```
USE AdventureWorks;
GO
ALTER FULLTEXT INDEX ON Production.Document
   SET SEARCH PROPERTY LIST OFF WITH NO POPULATION;
GO
```

D. Starting a full population

The following example starts a full population on the full-text index on the `JobCandidate` table of the `AdventureWorks` database.

```
USE AdventureWorks;
GO
ALTER FULLTEXT INDEX ON HumanResources.JobCandidate
   START FULL POPULATION;
GO
```

See Also

sys.fulltext_indexes (Transact-SQL)
CREATE FULLTEXT INDEX
DROP FULLTEXT INDEX
Full-Text Search
Full-Text Index Population

ALTER FULLTEXT STOPLIST

Inserts or deletes a stop word in the default full-text stoplist of the current database.

Important
CREATE FULLTEXT STOPLIST is supported only for compatibility level 100. For compatibility levels 80 and 90, the system stoplist is always assigned to the database.

Transact-SQL Syntax Conventions

Syntax

```
ALTER FULLTEXT STOPLIST stoplist_name
{
     ADD [N] 'stopword' LANGUAGE language_term
  | DROP
```

```
    {
       'stopword' LANGUAGE language_term
     | ALL LANGUAGE language_term
     | ALL
     }
;
```

Arguments

stoplist_name

Is the name of the stoplist being altered. stoplist_name can be a maximum of 128 characters.

'stopword'

Is a string that could be a word with linguistic meaning in the specified language or a token that does not have a linguistic meaning. stopword is limited to the maximum token length (64 characters). A stopword can be specified as a Unicode string.

LANGUAGE language_term

Specifies the language to associate with the stopword being added or dropped.

language_term can be specified as a string, integer, or hexadecimal value corresponding to the locale identifier (LCID) of the language, as follows:

Format	Description
String	language_term corresponds to the **alias** column value in the sys.syslanguages (Transact-SQL) compatibility view. The string must be enclosed in single quotation marks, as in 'language_term'.
Integer	language_term is the LCID of the language.
Hexadecimal	language_term is 0x followed by the hexadecimal value of the LCID. The hexadecimal value must not exceed eight digits, including leading zeros. If the value is in double-byte character set (DBCS) format, SQL Server converts it to Unicode.

ADD 'stopword' LANGUAGE language_term

Adds a stop word to the stoplist for the language specified by LANGUAGE language_term.

If the specified combination of keyword and the LCID value of the language is not unique in the STOPLIST, an error is returned. If the LCID value does not correspond to a registered language, an error is generated.

DROP { 'stopword' LANGUAGE language_term | ALL LANGUAGE language_term | ALL }

Drops a stop word from the stop list.

'stopword' LANGUAGE language_term

Drops the specified stop word for the language specified by language_term.

ALL LANGUAGE language_term

Drops all of the stop words for the language specified by language_term.

ALL

Drops all of the stop words in the stoplist.

Remarks

None.

Permissions

To designate a stoplist as the default stoplist of the database requires ALTER DATABASE permission. To otherwise alter a stoplist requires being the stoplist owner or membership in the **db_owner** or **db_ddladmin** fixed database roles.

Examples

The following example alters a stoplist named `CombinedFunctionWordList`, adding the word 'en', first for Spanish and then for French.

```
ALTER FULLTEXT STOPLIST CombinedFunctionWordList ADD 'en' LANGUAGE 'Spanish';
ALTER FULLTEXT STOPLIST CombinedFunctionWordList ADD 'en' LANGUAGE 'French';
```

See Also

CREATE FULLTEXT STOPLIST (Transact-SQL)

DROP FULLTEXT STOPLIST (Transact-SQL)

Noise Words

sys.fulltext_stoplists (Transact-SQL)

sys.fulltext_stopwords (Transact-SQL)

Configure and Manage Stopwords and Stoplists for Full-Text Search

ALTER FUNCTION

Alters an existing Transact-SQL or CLR function that was previously created by executing the CREATE FUNCTION statement, without changing permissions and without affecting any dependent functions, stored procedures, or triggers.

Transact-SQL Syntax Conventions

Syntax

Scalar Functions

```
ALTER FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ][ type_schema_name. ] parameter_data_type
    [ = default ] }
    [ ,...n ]
  ]
)
RETURNS return_data_type
    [ WITH <function_option> [ ,...n ] ]
    [ AS ]
    BEGIN
        function_body
        RETURN scalar_expression
    END
[ ; ]
```

Inline Table-valued Functions

```
ALTER FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ] [ type_schema_name. ] parameter_data_type
    [ = default ] }
    [ ,...n ]
  ]
)
RETURNS TABLE
    [ WITH <function_option> [ ,...n ] ]
    [ AS ]
    RETURN [ ( ] select_stmt [ ) ]
[ ; ]
```

Multistatement Table-valued Functions

```
ALTER FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ] [ type_schema_name. ] parameter_data_type
```

```
   [ = default ] }
   [ ,...n ]
 ]
)
RETURNS @return_variable TABLE <table_type_definition>
   [ WITH <function_option> [ ,...n ] ]
   [ AS ]
   BEGIN
      function_body
      RETURN
   END
[ ; ]
```

CLR Functions

```
ALTER FUNCTION [ schema_name. ] function_name
( { @parameter_name [AS] [ type_schema_name. ] parameter_data_type
   [ = default ] }
   [ ,...n ]
)
RETURNS { return_data_type | TABLE <clr_table_type_definition> }
   [ WITH <clr_function_option> [ ,...n ] ]
   [ AS ] EXTERNAL NAME <method_specifier>
[ ; ]
```

<method_specifier>::=

```
   assembly_name.class_name.method_name
```

Function Options

<function_option>::=

```
{
   [ ENCRYPTION ]
  | [ SCHEMABINDING ]
  | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
  | [ EXECUTE_AS_Clause ]
}
```

<clr_function_option>::=

}

[RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT]

| [**EXECUTE_AS_Clause**]

}

Table Type Definitions

<table_type_definition>:: =

({ **<column_definition> <column_constraint>**

| <computed_column_definition> }

[<table_constraint>] [**,...n**]

)

<clr_table_type_definition>:: =

({ **column_name data_type** } [**,...n**] **)**

<column_definition>::=

{

{ **column_name data_type** }

[[DEFAULT **constant_expression**]

[COLLATE **collation_name**] | [ROWGUIDCOL]

]

| [IDENTITY [**(seed , increment)**]]

[<column_constraint> [...**n**]]

}

<column_constraint>::=

{

[NULL | NOT NULL]

{ PRIMARY KEY | UNIQUE }

[CLUSTERED | NONCLUSTERED]

[WITH FILLFACTOR = **fillfactor**

| WITH **(** < index_option > [**,** ...n] **)**

[ON { filegroup | "default" }]

| [CHECK **(** **logical_expression** **)**] [**,...n**]

}

<computed_column_definition>::=

column_name AS **computed_column_expression**

<table_constraint>::=

{

{ PRIMARY KEY | UNIQUE }

[CLUSTERED | NONCLUSTERED]

(column_name [ASC | DESC] [**,...n**] **)**

[WITH FILLFACTOR = **fillfactor**

| WITH **(** <index_option> [**,** ...**n**] **)**

| [CHECK **(logical_expression))**] [**,...n**]

}

<index_option>::=

{

PAD_INDEX = { ON | OFF }

| FILLFACTOR = fillfactor

| IGNORE_DUP_KEY = { ON | OFF }

| STATISTICS_NORECOMPUTE = { ON | OFF }

| ALLOW_ROW_LOCKS = { ON | OFF }

| ALLOW_PAGE_LOCKS ={ ON | OFF }

}

Arguments

schema_name

Is the name of the schema to which the user-defined function belongs.

function_name

Is the user-defined function to be changed.

Note

Parentheses are required after the function name even if a parameter is not specified.

@parameter_name

Is a parameter in the user-defined function. One or more parameters can be declared.

A function can have a maximum of 2,100 parameters. The value of each declared parameter must be supplied by the user when the function is executed, unless a default for the

parameter is defined.

Specify a parameter name by using an at sign (**@**) as the first character. The parameter name must comply with the rules for identifiers. Parameters are local to the function; the same parameter names can be used in other functions. Parameters can take the place only of constants; they cannot be used instead of table names, column names, or the names of other database objects.

Note

ANSI_WARNINGS is not honored when passing parameters in a stored 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.

[type_schema_name.] parameter_data_type

Is the parameter data type and optionally, the schema to which it belongs. For Transact-SQL functions, all data types, including CLR user-defined types, are allowed except the **timestamp** data type. For CLR functions, all data types, including CLR user-defined types, are allowed except **text**, **ntext**, **image**, and **timestamp** data types. The nonscalar types **cursor** and **table** cannot be specified as a parameter data type in either Transact-SQL or CLR functions.

If type_schema_name is not specified, the SQL Server 2005 Database Engine looks for the parameter_data_type in the following order:

- The schema that contains the names of SQL Server system data types.
- The default schema of the current user in the current database.
- The **dbo** schema in the current database.

[= default]

Is a default value for the parameter. If a default value is defined, the function can be executed without specifying a value for that parameter.

Note

Default parameter values can be specified for CLR functions except for **varchar(max)** and **varbinary(max)** data types.

When a parameter of the function has a default value, the keyword DEFAULT must be specified when calling the function to retrieve the default value. This behavior is different from using parameters with default values in stored procedures in which omitting the parameter also implies the default value.

return_data_type

Is the return value of a scalar user-defined function. For Transact-SQL functions, all data types, including CLR user-defined types, are allowed except the **timestamp** data type. For CLR functions, all data types, including CLR user-defined types, are allowed except **text**, **ntext**, **image**, and **timestamp** data types. The nonscalar types **cursor** and **table** cannot be specified as a return data type in either Transact-SQL or CLR functions.

function_body

Specifies that a series of Transact-SQL statements, which together do not produce a side effect such as modifying a table, define the value of the function. function_body is used only in scalar functions and multistatement table-valued functions.

In scalar functions, function_body is a series of Transact-SQL statements that together evaluate to a scalar value.

In multistatement table-valued functions, function_body is a series of Transact-SQL statements that populate a TABLE return variable.

scalar_expression

Specifies that the scalar function returns a scalar value.

TABLE

Specifies that the return value of the table-valued function is a table. Only constants and **@**local_variables can be passed to table-valued functions.

In inline table-valued functions, the TABLE return value is defined through a single SELECT statement. Inline functions do not have associated return variables.

In multistatement table-valued functions, **@**return_variable is a TABLE variable used to store and accumulate the rows that should be returned as the value of the function. **@**return_variable can be specified only for Transact-SQL functions and not for CLR functions.

select-stmt

Is the single SELECT statement that defines the return value of an inline table-valued function.

EXTERNAL NAME <method_specifier>assembly_name.class_name.method_name

Specifies the method of an assembly to bind with the function. assembly_name must match an existing assembly in SQL Server in the current database with visibility on. 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 (**" "**). method_name must be a valid SQL Server identifier and must exist as a static method in the specified class.

Note

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

Note

This option is not available in a contained database.

<table_type_definition>({ <column_definition> <column_constraint> | <computed_column_definition> } [<table_constraint>] [,...n])

Defines the table data type for a Transact-SQL function. The table declaration includes column definitions and column or table constraints.

< clr_table_type_definition > ({ column_name data_type } [,...n])

Defines the table data types for a CLR function. The table declaration includes only column names and data types.

<function_option>::= and <clr_function_option>::=

Specifies the function will have one or more of the following options.

ENCRYPTION

Indicates that the Database Engine encrypts the catalog view columns that contains the text of the ALTER FUNCTION statement. Using ENCRYPTION prevents the function from being published as part of SQL Server replication. ENCRYPTION cannot be specified for CLR functions.

SCHEMABINDING

Specifies that the function is bound to the database objects that it references. This condition will prevent changes to the function if other schema bound objects are referencing it.

The binding of the function to the objects it references is removed only when one of the following actions occurs:

- The function is dropped.
- The function is modified by using the ALTER statement with the SCHEMABINDING option not specified.

For a list of conditions that must be met before a function can be schema bound, see EVENTDATA (Transact-SQL).

RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT

Specifies the **OnNULLCall** attribute of a scalar-valued function. If not specified, CALLED ON NULL INPUT is implied by default. This means that the function body executes even if NULL is passed as an argument.

If RETURNS NULL ON NULL INPUT is specified in a CLR function, it indicates that SQL Server can return NULL when any of the arguments it receives is NULL, without actually invoking the body of the function. If the method specified in <method_specifier> already has a custom attribute that indicates RETURNS NULL ON NULL INPUT, but the ALTER FUNCTION statement indicates CALLED ON NULL INPUT, the ALTER FUNCTION statement takes precedence. The **OnNULLCall** attribute cannot be specified for CLR table-valued functions.

EXECUTE AS Clause

Specifies the security context under which the user-defined function is executed. Therefore, you can control which user account SQL Server uses to validate permissions on any database objects referenced by the function.

Note

EXECUTE AS cannot be specified for inline user-defined functions.

For more information, see EXECUTE AS Clause (Transact-SQL).

< column_definition >::=

Defines the table data type. The table declaration includes column definitions and constraints. For CLR functions, only column_name and data_type can be specified.

column_name

Is the name of a column in the table. Column names must comply with the rules for identifiers and must be unique in the table. column_name can consist of 1 through 128 characters.

data_type

Specifies the column data type. For Transact-SQL functions, all data types, including CLR user-defined types, are allowed except **timestamp**. For CLR functions, all data types, including CLR user-defined types, are allowed except **text**, **ntext**, **image**, **char**, **varchar**, **varchar(max)**, and **timestamp**.The nonscalar type **cursor** cannot be specified as a column data type in either Transact-SQL or CLR functions.

DEFAULT constant_expression

Specifies the value provided for the column when a value is not explicitly supplied during an insert. constant_expression is a constant, NULL, or a system function value. DEFAULT definitions can be applied to any column except those that have the IDENTITY property. DEFAULT cannot be specified for CLR table-valued functions.

COLLATE collation_name

Specifies the collation for the column. If not specified, the column is assigned the default collation of the database. Collation name can be either a Windows collation name or a SQL collation name. For a list of and more information, see Windows Collation Name and SQL Collation Name.

The COLLATE clause can be used to change the collations only of columns of the **char**, **varchar**, **nchar**, and **nvarchar** data types.

COLLATE cannot be specified for CLR table-valued functions.

ROWGUIDCOL

Indicates that the new column is a row global unique identifier column. Only one **uniqueidentifier** column per table can be designated as the ROWGUIDCOL column. The ROWGUIDCOL property can be assigned only to a **uniqueidentifier** column.

The ROWGUIDCOL property does not enforce uniqueness of the values stored in the column. It also does not automatically generate values for new rows inserted into the table. To generate unique values for each column, use the NEWID function on INSERT statements. A default value can be specified; however, NEWID cannot be specified as the default.

IDENTITY

Indicates that the new column is an identity column. When a new row is added to the table, SQL Server provides a unique, incremental value for the column. Identity columns are typically used together with PRIMARY KEY constraints to serve as the unique row identifier for the table. The IDENTITY property can be assigned to **tinyint**, **smallint**, **int**, **bigint**, **decimal(p,0)**, or **numeric(p,0)** columns. Only one identity column can be created per table. Bound defaults and DEFAULT constraints cannot be used with an identity column. You must specify both the seed and increment or neither. If neither is specified, the default is (1,1).

IDENTITY cannot be specified for CLR table-valued functions.

seed

Is the integer value to be assigned to the first row in the table.

increment

Is the integer value to add to the seed value for successive rows in the table.

< column_constraint >::= and < table_constraint>::=

Defines the constraint for a specified column or table. For CLR functions, the only constraint type allowed is NULL. Named constraints are not allowed.

NULL | NOT NULL

Determines whether null values are allowed in the column. NULL is not strictly a constraint but can be specified just like NOT NULL. NOT NULL cannot be specified for CLR table-valued functions.

PRIMARY KEY

Is a constraint that enforces entity integrity for a specified column through a unique index. In table-valued user-defined functions, the PRIMARY KEY constraint can be created on only one column per table. PRIMARY KEY cannot be specified for CLR table-valued functions.

UNIQUE

Is a constraint that provides entity integrity for a specified column or columns through a unique index. A table can have multiple UNIQUE constraints. UNIQUE cannot be specified for CLR table-valued functions.

CLUSTERED | NONCLUSTERED

Indicate that a clustered or a nonclustered index is created for the PRIMARY KEY or UNIQUE constraint. PRIMARY KEY constraints use CLUSTERED, and UNIQUE constraints use NONCLUSTERED.

CLUSTERED can be specified for only one constraint. If CLUSTERED is specified for a UNIQUE constraint and a PRIMARY KEY constraint is also specified, the PRIMARY KEY uses NONCLUSTERED.

CLUSTERED and NONCLUSTERED cannot be specified for CLR table-valued functions.

CHECK

Is a constraint that enforces domain integrity by limiting the possible values that can be entered into a column or columns. CHECK constraints cannot be specified for CLR table-valued functions.

logical_expression

Is a logical expression that returns TRUE or FALSE.

<computed_column_definition>::=

Specifies a computed column. For more information about computed columns, see CREATE TABLE (Transact-SQL).

column_name

Is the name of the computed column.

computed_column_expression

Is an expression that defines the value of a computed column.

<index_option>::=

Specifies the index options for the PRIMARY KEY or UNIQUE index. For more information about index options, see CREATE INDEX (Transact-SQL).

PAD_INDEX = { ON | OFF }

Specifies index padding. The default is OFF.

FILLFACTOR = fillfactor

Specifies a percentage that indicates how full the Database Engine should make the leaf level of each index page during index creation or change. fillfactor must be an integer value from 1 to 100. The default is 0.

IGNORE_DUP_KEY = { ON | OFF }

Specifies the error response when an insert operation attempts to insert duplicate key values into a unique index. The IGNORE_DUP_KEY option applies only to insert operations after the index is created or rebuilt. The default is OFF.

STATISTICS_NORECOMPUTE = { ON | OFF }

Specifies whether distribution statistics are recomputed. The default is OFF.

ALLOW_ROW_LOCKS = { ON | OFF }

Specifies whether row locks are allowed. The default is ON.

ALLOW_PAGE_LOCKS = { ON | OFF }

Specifies whether page locks are allowed. The default is ON.

Remarks

ALTER FUNCTION cannot be used to change a scalar-valued function to a table-valued function, or vice versa. Also, ALTER FUNCTION cannot be used to change an inline function to a

multistatement function, or vice versa. ALTER FUNCTION cannot be used to change a Transact-SQL function to a CLR function or vice-versa.

The following Service Broker statements cannot be included in the definition of a Transact-SQL user-defined function:

- BEGIN DIALOG CONVERSATION
- END CONVERSATION
- GET CONVERSATION GROUP
- MOVE CONVERSATION
- RECEIVE
- SEND

Permissions

Requires ALTER permission on the function or on the schema. If the function specifies a user-defined type, requires EXECUTE permission on the type.

See Also

CREATE FUNCTION (Transact-SQL)

DROP FUNCTION (Transact-SQL)

Making Schema Changes on Publication Databases

EVENTDATA (Transact-SQL)

ALTER INDEX

Modifies an existing table or view index (relational or XML) by disabling, rebuilding, or reorganizing the index; or by setting options on the index.

 Transact-SQL Syntax Conventions

Syntax

```
ALTER INDEX { index_name | ALL }
   ON <object>
   { REBUILD
      [ [PARTITION = ALL]
       [ WITH ( <rebuild_index_option> [ ,...n ] ) ]
       | [ PARTITION = partition_number
            [ WITH ( <single_partition_rebuild_index_option>
                 [ ,...n ] )
            ]
        ]
```

```
      ]
   | DISABLE
   | REORGANIZE
      [ PARTITION = partition_number ]
      [ WITH ( LOB_COMPACTION = { ON | OFF } ) ]
  | SET ( <set_index_option> [ ,...n ] )
   }
[ ; ]

<object> ::=
{
   [ database_name. [ schema_name ] . | schema_name. ]
   table_or_view_name
}

<rebuild_index_option > ::=
{
   PAD_INDEX = { ON | OFF }
  | FILLFACTOR = fillfactor
  | SORT_IN_TEMPDB = { ON | OFF }
  | IGNORE_DUP_KEY = { ON | OFF }
  | STATISTICS_NORECOMPUTE = { ON | OFF }
  | ONLINE = { ON | OFF }
  | ALLOW_ROW_LOCKS = { ON | OFF }
  | ALLOW_PAGE_LOCKS = { ON | OFF }
  | MAXDOP = max_degree_of_parallelism
  | DATA_COMPRESSION = { NONE | ROW | PAGE }
    [ ON PARTITIONS ( { <partition_number_expression> | <range> }
    [ , ...n ] ) ]
}
<range> ::=
<partition_number_expression> TO <partition_number_expression>
}

<single_partition_rebuild_index_option> ::=
```

```
{
   SORT_IN_TEMPDB = { ON | OFF }
  | MAXDOP = max_degree_of_parallelism
  | DATA_COMPRESSION = { NONE | ROW | PAGE } }
}
```

<set_index_option>::=

```
{
   ALLOW_ROW_LOCKS = { ON | OFF }
  | ALLOW_PAGE_LOCKS = { ON | OFF }
  | IGNORE_DUP_KEY = { ON | OFF }
  | STATISTICS_NORECOMPUTE = { ON | OFF }
}
```

Arguments

index_name

Is the name of the index. Index names must be unique within a table or view but do not have to be unique within a database. Index names must follow the rules of identifiers.

ALL

Specifies all indexes associated with the table or view regardless of the index type. Specifying ALL causes the statement to fail if one or more indexes are in an offline or read-only filegroup or the specified operation is not allowed on one or more index types. The following table lists the index operations and disallowed index types.

Specifying ALL with this operation	Fails if the table has one or more
REBUILD WITH ONLINE = ON	XML index Spatial index Large object data type columns: **image**, **text**, **ntext**, **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, and **xml**
REBUILD PARTITION = partition_number	Nonpartitioned index, XML index, spatial index, or disabled index
REORGANIZE	Indexes with ALLOW_PAGE_LOCKS set to OFF
REORGANIZE PARTITION = partition_number	Nonpartitioned index, XML index, spatial index, or disabled index

IGNORE_DUP_KEY = ON	Spatial index XML index
ONLINE = ON	Spatial index XML index

If ALL is specified with PARTITION = partition_number, all indexes must be aligned. This means that they are partitioned based on equivalent partition functions. Using ALL with PARTITION causes all index partitions with the same partition_number to be rebuilt or reorganized. For more information about partitioned indexes, see Partitioned Tables and Indexes.

database_name

Is the name of the database.

schema_name

Is the name of the schema to which the table or view belongs.

table_or_view_name

Is the name of the table or view associated with the index. To display a report of the indexes on an object, use the sys.indexes catalog view.

REBUILD [WITH (<rebuild_index_option> [,... n])]

Specifies the index will be rebuilt using the same columns, index type, uniqueness attribute, and sort order. This clause is equivalent to DBCC DBREINDEX. REBUILD enables a disabled index. Rebuilding a clustered index does not rebuild associated nonclustered indexes unless the keyword ALL is specified. If index options are not specified, the existing index option values stored in sys.indexes are applied. For any index option whose value is not stored in **sys.indexes**, the default indicated in the argument definition of the option applies.

When you rebuild an XML index or a spatial index, the options ONLINE = ON and IGNORE_DUP_KEY = ON are not valid.

If ALL is specified and the underlying table is a heap, the rebuild operation has no effect on the table. Any nonclustered indexes associated with the table are rebuilt.

The rebuild operation can be minimally logged if the database recovery model is set to either bulk-logged or simple.

Note

When you rebuild a primary XML index, the underlying user table is unavailable for the duration of the index operation.

PARTITION

Specifies that only one partition of an index will be rebuilt or reorganized. PARTITION cannot

be specified if index_name is not a partitioned index.

PARTITION = ALL rebuilds all partitions.

Warning

Creating and rebuilding nonaligned indexes on a table with more than 1,000 partitions is possible, but is not supported. Doing so may cause degraded performance or excessive memory consumption during these operations. We recommend using only aligned indexes when the number of partitions exceed 1,000.

partition_number

Is the partition number of a partitioned index that is to be rebuilt or reorganized. partition_number is a constant expression that can reference variables. These include user-defined type variables or functions and user-defined functions, but cannot reference a Transact-SQL statement. partition_number must exist or the statement fails.

WITH (<single_partition_rebuild_index_option>)

SORT_IN_TEMPDB, MAXDOP, and DATA_COMPRESSION are the options that can be specified when you rebuild a single partition (PARTITION = n). XML indexes cannot be specified in a single partition rebuild operation.

Rebuilding a partitioned index cannot be performed online. The entire table is locked during this operation.

DISABLE

Marks the index as disabled and unavailable for use by the Database Engine. Any index can be disabled. The index definition of a disabled index remains in the system catalog with no underlying index data. Disabling a clustered index prevents user access to the underlying table data. To enable an index, use ALTER INDEX REBUILD or CREATE INDEX WITH DROP_EXISTING. For more information, see Disable Indexes and Constraints and Enable Indexes and Constraints.

REORGANIZE

Specifies the index leaf level will be reorganized. ALTER INDEX REORGANIZE statement is always performed online. This means long-term blocking table locks are not held and queries or updates to the underlying table can continue during the ALTER INDEX REORGANIZE transaction. REORGANIZE cannot be specified for a disabled index or an index with ALLOW_PAGE_LOCKS set to OFF.

WITH (LOB_COMPACTION = { ON | OFF })

Specifies that all pages that contain large object (LOB) data are compacted. The LOB data types are **image**, **text**, **ntext**, **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, and **xml**. Compacting this data can improve disk space use. The default is ON.

ON

All pages that contain large object data are compacted.

Reorganizing a specified clustered index compacts all LOB columns that are contained in

the clustered index. Reorganizing a nonclustered index compacts all LOB columns that are nonkey (included) columns in the index.

When ALL is specified, all indexes that are associated with the specified table or view are reorganized, and all LOB columns that are associated with the clustered index, underlying table, or nonclustered index with included columns are compacted.

OFF

Pages that contain large object data are not compacted.

OFF has no effect on a heap.

The LOB_COMPACTION clause is ignored if LOB columns are not present.

SET (<set_index option> [,... n])

Specifies index options without rebuilding or reorganizing the index. SET cannot be specified for a disabled index.

PAD_INDEX = { ON | OFF }

Specifies index padding. The default is OFF.

ON

The percentage of free space that is specified by FILLFACTOR is applied to the intermediate-level pages of the index. If FILLFACTOR is not specified at the same time PAD_INDEX is set to ON, the fill factor value stored in sys.indexes is used.

OFF or fillfactor is not specified

The intermediate-level pages are filled to near capacity. This leaves sufficient space for at least one row of the maximum size that the index can have, based on the set of keys on the intermediate pages.

For more information, see CREATE INDEX (Transact-SQL).

FILLFACTOR = fillfactor

Specifies a percentage that indicates how full the Database Engine should make the leaf level of each index page during index creation or alteration. fillfactor must be an integer value from 1 to 100. The default is 0.

Note

Fill factor values 0 and 100 are the same in all respects.

An explicit FILLFACTOR setting applies only when the index is first created or rebuilt. The Database Engine does not dynamically keep the specified percentage of empty space in the pages. For more information, see CREATE INDEX.

To view the fill factor setting, use **sys.indexes**.

Important

Creating or altering a clustered index with a FILLFACTOR value affects the amount of storage space the data occupies, because the Database Engine redistributes the data when it creates the clustered

index.

SORT_IN_TEMPDB = { ON | OFF }

Specifies whether to store the sort results in **tempdb**. The default is OFF.

ON

The intermediate sort results that are used to build the index are stored in **tempdb**. If **tempdb** is on a different set of disks than the user database, this may reduce the time needed to create an index. However, this increases the amount of disk space that is used during the index build.

OFF

The intermediate sort results are stored in the same database as the index.

If a sort operation is not required, or if the sort can be performed in memory, the SORT_IN_TEMPDB option is ignored.

For more information, see tempdb and Index Creation.

IGNORE_DUP_KEY = { ON | OFF }

Specifies the error response when an insert operation attempts to insert duplicate key values into a unique index. The IGNORE_DUP_KEY option applies only to insert operations after the index is created or rebuilt. The default is OFF.

ON

A warning message will occur when duplicate key values are inserted into a unique index. Only the rows violating the uniqueness constraint will fail.

OFF

An error message will occur when duplicate key values are inserted into a unique index. The entire INSERT operation will be rolled back.

IGNORE_DUP_KEY cannot be set to ON for indexes created on a view, non-unique indexes, XML indexes, spatial indexes, and filtered indexes.

To view IGNORE_DUP_KEY, use sys.indexes.

In backward compatible syntax, WITH IGNORE_DUP_KEY is equivalent to WITH IGNORE_DUP_KEY = ON.

STATISTICS_NORECOMPUTE = { ON | OFF }

Specifies whether distribution statistics are recomputed. The default is OFF.

ON

Out-of-date statistics are not automatically recomputed.

OFF

Automatic statistics updating are enabled.

To restore automatic statistics updating, set the STATISTICS_NORECOMPUTE to OFF, or execute UPDATE STATISTICS without the NORECOMPUTE clause.

Important

Disabling automatic recomputation of distribution statistics may prevent the query optimizer from picking optimal execution plans for queries that involve the table.

ONLINE = { ON | OFF }

Specifies whether underlying tables and associated indexes are available for queries and data modification during the index operation. The default is OFF.

For an XML index or spatial index, only ONLINE = OFF is supported, and if ONLINE is set to ON an error is raised.

Note

Online index operations are not available in every edition of Microsoft SQL Server. For a list of features that are supported by the editions of SQL Server, see Features Supported by the Editions of SQL Server 2012.

ON

Long-term table locks are not held for the duration of the index operation. During the main phase of the index operation, only an Intent Share (IS) lock is held on the source table. This allows queries or updates to the underlying table and indexes to continue. At the start of the operation, a Shared (S) lock is very briefly held on the source object. At the end of the operation, an S lock is very briefly held on the source if a nonclustered index is being created, or an SCH-M (Schema Modification) lock is acquired when a clustered index is created or dropped online, or when a clustered or nonclustered index is being rebuilt. ONLINE cannot be set to ON when an index is being created on a local temporary table.

OFF

Table locks are applied for the duration of the index operation. An offline index operation that creates, rebuilds, or drops a clustered, spatial, or XML index, or rebuilds or drops a nonclustered index, acquires a Schema modification (Sch-M) lock on the table. This prevents all user access to the underlying table for the duration of the operation. An offline index operation that creates a nonclustered index acquires a Shared (S) lock on the table. This prevents updates to the underlying table but allows read operations, such as SELECT statements.

For more information, see How Online Index Operations Work.

Indexes, including indexes on global temp tables, can be rebuilt online with the following exceptions:

- XML indexes
- Indexes on local temp tables
- A subset of a partitioned index (An entire partitioned index can be rebuilt online.)
- Clustered indexes if the underlying table contains LOB data types
- Nonclustered indexes that are defined with the **image**, **ntext**, and **text** data type columns

Nonclustered indexes can be rebuilt online if the table contains LOB data types but none of these columns are used in the index definition as either key or nonkey columns.

ALLOW_ROW_LOCKS = { ON | OFF }

Specifies whether row locks are allowed. The default is ON.

ON

Row locks are allowed when accessing the index. The Database Engine determines when row locks are used.

OFF

Row locks are not used.

ALLOW_PAGE_LOCKS = { ON | OFF }

Specifies whether page locks are allowed. The default is ON.

ON

Page locks are allowed when you access the index. The Database Engine determines when page locks are used.

OFF

Page locks are not used.

Note

An index cannot be reorganized when ALLOW_PAGE_LOCKS is set to OFF.

MAXDOP = max_degree_of_parallelism

Overrides the **max degree of parallelism** configuration option for the duration of the index operation. For more information, see Configure the max degree of parallelism Server Configuration Option. Use MAXDOP to limit the number of processors used in a parallel plan execution. The maximum is 64 processors.

Important

Although the MAXDOP option is syntactically supported for all XML indexes, for a spatial index or a primary XML index, ALTER INDEX currently uses only a single processor.

max_degree_of_parallelism can be:

1

Suppresses parallel plan generation.

>1

Restricts the maximum number of processors used in a parallel index operation to the specified number.

0 (default)

Uses the actual number of processors or fewer based on the current system workload.

For more information, see Configuring Parallel Index Operations.

Note

Parallel index operations are not available in every edition of Microsoft SQL Server. For a list of features that are supported by the editions of SQL Server, see Features Supported by the Editions of SQL Server 2012.

DATA_COMPRESSION

Specifies the data compression option for the specified index, partition number, or range of partitions. The options are as follows:

NONE

Index or specified partitions are not compressed.

ROW

Index or specified partitions are compressed by using row compression.

PAGE

Index or specified partitions are compressed by using page compression.

For more information about compression, see Creating Compressed Tables and Indexes.

ON PARTITIONS ({ <partition_number_expression> | <range> } [,...n])

Specifies the partitions to which the DATA_COMPRESSION setting applies. If the index is not partitioned, the ON PARTITIONS argument will generate an error. If the ON PARTITIONS clause is not provided, the DATA_COMPRESSION option applies to all partitions of a partitioned index.

<partition_number_expression> can be specified in the following ways:

- Provide the number for a partition, for example: ON PARTITIONS (2).
- Provide the partition numbers for several individual partitions separated by commas, for example: ON PARTITIONS (1, 5).
- Provide both ranges and individual partitions: ON PARTITIONS (2, 4, 6 TO 8).

<range> can be specified as partition numbers separated by the word TO, for example: ON PARTITIONS (6 TO 8).

To set different types of data compression for different partitions, specify the DATA_COMPRESSION option more than once, for example:

```
REBUILD WITH
(
DATA_COMPRESSION = NONE ON PARTITIONS (1),
DATA_COMPRESSION = ROW ON PARTITIONS (2, 4, 6 TO 8),
DATA_COMPRESSION = PAGE ON PARTITIONS (3, 5)
)
```

Remarks

ALTER INDEX cannot be used to repartition an index or move it to a different filegroup. This statement cannot be used to modify the index definition, such as adding or deleting columns or changing the column order. Use CREATE INDEX with the DROP_EXISTING clause to perform these operations.

When an option is not explicitly specified, the current setting is applied. For example, if a FILLFACTOR setting is not specified in the REBUILD clause, the fill factor value stored in the system catalog will be used during the rebuild process. To view the current index option settings, use sys.indexes.

Note

The values for ONLINE, MAXDOP, and SORT_IN_TEMPDB are not stored in the system catalog. Unless specified in the index statement, the default value for the option is used.

On multiprocessor computers, just like other queries do, ALTER INDEX REBUILD automatically uses more processors to perform the scan and sort operations that are associated with modifying the index. When you run ALTER INDEX REORGANIZE, with or without LOB_COMPACTION, the **max degree of parallelism** value is a single threaded operation. For more information, see Configuring Parallel Index Operations.

An index cannot be reorganized or rebuilt if the filegroup in which it is located is offline or set to read-only. When the keyword ALL is specified and one or more indexes are in an offline or read-only filegroup, the statement fails.

Rebuilding Indexes

Rebuilding an index drops and re-creates the index. This removes fragmentation, reclaims disk space by compacting the pages based on the specified or existing fill factor setting, and reorders the index rows in contiguous pages. When ALL is specified, all indexes on the table are dropped and rebuilt in a single transaction. FOREIGN KEY constraints do not have to be dropped in advance. When indexes with 128 extents or more are rebuilt, the Database Engine defers the actual page deallocations, and their associated locks, until after the transaction commits.

Rebuilding or reorganizing small indexes often does not reduce fragmentation. The pages of small indexes are stored on mixed extents. Mixed extents are shared by up to eight objects, so the fragmentation in a small index might not be reduced after reorganizing or rebuilding it.

In SQL Server 2012, statistics are not created by scanning all the rows in the table when a partitioned index is created or rebuilt. Instead, the query optimizer uses the default sampling algorithm to generate statistics. To obtain statistics on partitioned indexes by scanning all the rows in the table, use CREATE STATISTICS or UPDATE STATISTICS with the FULLSCAN clause.

In earlier versions of SQL Server, you could sometimes rebuild a nonclustered index to correct inconsistencies caused by hardware failures. In SQL Server 2008 and later, you may still be able to repair such inconsistencies between the index and the clustered index by rebuilding a nonclustered index offline. However, you cannot repair nonclustered index inconsistencies by rebuilding the index online, because the online rebuild mechanism will use the existing nonclustered index as the basis for the rebuild and thus persist the inconsistency. Rebuilding the

index offline, by contrast, will force a scan of the clustered index (or heap) and so remove the inconsistency. As with earlier versions, we recommend recovering from inconsistencies by restoring the affected data from a backup; however, you may be able to repair the index inconsistencies by rebuilding the nonclustered index offline. For more information, see DBCC CHECKDB (Transact-SQL).

Reorganizing Indexes

Reorganizing an index uses minimal system resources. It defragments the leaf level of clustered and nonclustered indexes on tables and views by physically reordering the leaf-level pages to match the logical, left to right, order of the leaf nodes. Reorganizing also compacts the index pages. Compaction is based on the existing fill factor value. To view the fill factor setting, use sys.indexes.

When ALL is specified, relational indexes, both clustered and nonclustered, and XML indexes on the table are reorganized. Some restrictions apply when specifying ALL, see the definition for ALL in the Arguments section.

For more information, see Reorganizing and Rebuilding Indexes.

Disabling Indexes

Disabling an index prevents user access to the index, and for clustered indexes, to the underlying table data. The index definition remains in the system catalog. Disabling a nonclustered index or clustered index on a view physically deletes the index data. Disabling a clustered index prevents access to the data, but the data remains unmaintained in the B-tree until the index is dropped or rebuilt. To view the status of an enabled or disabled index, query the **is_disabled** column in the **sys.indexes** catalog view.

If a table is in a transactional replication publication, you cannot disable any indexes that are associated with primary key columns. These indexes are required by replication. To disable an index, you must first drop the table from the publication. For more information, see Publishing Data and Database Objects.

Use the ALTER INDEX REBUILD statement or the CREATE INDEX WITH DROP_EXISTING statement to enable the index. Rebuilding a disabled clustered index cannot be performed with the ONLINE option set to ON. For more information, see Disable Indexes and Constraints.

Setting Options

You can set the options ALLOW_ROW_LOCKS, ALLOW_PAGE_LOCKS, IGNORE_DUP_KEY and STATISTICS_NORECOMPUTE for a specified index without rebuilding or reorganizing that index. The modified values are immediately applied to the index. To view these settings, use **sys.indexes**. For more information, see Setting Index Options.

Row and Page Locks Options

When ALLOW_ROW_LOCKS = ON and ALLOW_PAGE_LOCK = ON, row-level, page-level, and table-level locks are allowed when you access the index. The Database Engine chooses the appropriate lock and can escalate the lock from a row or page lock to a table lock.

When ALLOW_ROW_LOCKS = OFF and ALLOW_PAGE_LOCK = OFF, only a table-level lock is allowed when you access the index.

If ALL is specified when the row or page lock options are set, the settings are applied to all indexes. When the underlying table is a heap, the settings are applied in the following ways:

ALLOW_ROW_LOCKS = ON or OFF	To the heap and any associated nonclustered indexes.
ALLOW_PAGE_LOCKS = ON	To the heap and any associated nonclustered indexes.
ALLOW_PAGE_LOCKS = OFF	Fully to the nonclustered indexes. This means that all page locks are not allowed on the nonclustered indexes. On the heap, only the shared (S), update (U) and exclusive (X) locks for the page are not allowed. The Database Engine can still acquire an intent page lock (IS, IU or IX) for internal purposes.

Online Index Operations

When rebuilding an index and the ONLINE option is set to ON, the underlying objects, the tables and associated indexes, are available for queries and data modification. Exclusive table locks are held only for a very short amount of time during the alteration process.

Reorganizing an index is always performed online. The process does not hold locks long term and, therefore, does not block queries or updates that are running.

You can perform concurrent online index operations on the same table only when doing the following:

- Creating multiple nonclustered indexes.
- Reorganizing different indexes on the same table.
- Reorganizing different indexes while rebuilding nonoverlapping indexes on the same table.

All other online index operations performed at the same time fail. For example, you cannot rebuild two or more indexes on the same table concurrently, or create a new index while rebuilding an existing index on the same table.

For more information, see Performing Index Operations Online.

Spatial Index Restrictions

When you rebuild a spatial index, the underlying user table is unavailable for the duration of the index operation because the spatial index holds a schema lock.

The PRIMARY KEY constraint in the user table cannot be modified while a spatial index is defined on a column of that table. To change the PRIMARY KEY constraint, first drop every spatial index of the table. After modifying the PRIMARY KEy constraint, you can re-create each of the spatial indexes.

In a single partition rebuild operation, you cannot specify any spatial indexes. However, you can specify spatial indexes in a complete partition rebuild.

To change options that are specific to a spatial index, such as BOUNDING_BOX or GRID, you can either use a CREATE SPATIAL INDEX statement that specifies DROP_EXISTING = ON, or drop the spatial index and create a new one. For an example, see CREATE SPATIAL INDEX (Transact-SQL).

Columnstore Index Restrictions

Except for the REBUILD option, an xVelocity memory optimized columnstore index cannot be altered. Drop and recreate the columnstore index instead.

Data Compression

For a more information about data compression, see Creating Compressed Tables and Indexes.

To evaluate how changing the compression state will affect a table, an index, or a partition, use the sp_estimate_data_compression_savings stored procedure.

The following restrictions apply to partitioned indexes:

- When you use ALTER INDEX ALL ..., you cannot change the compression setting of a single partition if the table has nonaligned indexes.
- The ALTER INDEX <index> ... REBUILD PARTITION ... syntax rebuilds the specified partition of the index.
- The ALTER INDEX <index> ... REBUILD WITH ... syntax rebuilds all partitions of the index.

Statistics

When you execute **ALTER INDEX ALL ...** on a table, only the statistics associates with indexes are updated. Automatic or manual statistics created on the table (instead of an index) are not updated.

Permissions

To execute ALTER INDEX, at a minimum, ALTER permission on the table or view is required.

Examples

A. Rebuilding an index

The following example rebuilds a single index on the `Employee` table.

```
USE AdventureWorks2012;
GO
ALTER INDEX PK_Employee_BusinessEntityID ON HumanResources.Employee
REBUILD;
GO
```

B. Rebuilding all indexes on a table and specifying options

The following example specifies the keyword ALL. This rebuilds all indexes associated with the table. Three options are specified.

```
USE AdventureWorks2012;
GO
ALTER INDEX ALL ON Production.Product
REBUILD WITH (FILLFACTOR = 80, SORT_IN_TEMPDB = ON,
              STATISTICS_NORECOMPUTE = ON);
GO
```

C. Reorganizing an index with LOB compaction

The following example reorganizes a single clustered index. Because the index contains a LOB data type in the leaf level, the statement also compacts all pages that contain the large object data. Note that specifying the WITH (LOB_COMPACTION) option is not required because the default value is ON.

```
USE AdventureWorks2012;
GO
ALTER INDEX PK_ProductPhoto_ProductPhotoID ON Production.ProductPhoto
REORGANIZE ;
GO
```

D. Setting options on an index

The following example sets several options on the index AK_SalesOrderHeader_SalesOrderNumber.

```
USE AdventureWorks2012;
GO
ALTER INDEX AK_SalesOrderHeader_SalesOrderNumber ON
    Sales.SalesOrderHeader
SET (
    STATISTICS_NORECOMPUTE = ON,
    IGNORE_DUP_KEY = ON,
    ALLOW_PAGE_LOCKS = ON
    ) ;
GO
```

E. Disabling an index

The following example disables a nonclustered index on the **Employee** table.

```
USE AdventureWorks2012;
GO
ALTER INDEX IX_Employee_OrganizationNode ON HumanResources.Employee
DISABLE ;
GO
```

F. Disabling constraints

The following example disables a PRIMARY KEY constraint by disabling the PRIMARY KEY index. The FOREIGN KEY constraint on the underlying table is automatically disabled and warning message is displayed.

```
USE AdventureWorks2012;
GO
ALTER INDEX PK_Department_DepartmentID ON HumanResources.Department
DISABLE ;
GO
```

The result set returns this warning message.

```
Warning: Foreign key 'FK_EmployeeDepartmentHistory_Department_DepartmentID'
on table 'EmployeeDepartmentHistory' referencing table 'Department'
was disabled as a result of disabling the index 'PK_Department_DepartmentID'.
```

G. Enabling constraints

The following example enables the PRIMARY KEY and FOREIGN KEY constraints that were disabled in Example F.

The PRIMARY KEY constraint is enabled by rebuilding the PRIMARY KEY index.

```
USE AdventureWorks2012;
GO
ALTER INDEX PK_Department_DepartmentID ON HumanResources.Department
REBUILD ;
GO
```

The FOREIGN KEY constraint is then enabled.

```
ALTER TABLE HumanResources.EmployeeDepartmentHistory
CHECK CONSTRAINT FK_EmployeeDepartmentHistory_Department_DepartmentID;
GO
```

H. Rebuilding a partitioned index

The following example rebuilds a single partition, partition number 5, of the partitioned index IX_TransactionHistory_TransactionDate.

```
USE AdventureWorks;
GO
-- Verify the partitioned indexes.
SELECT *
FROM sys.dm_db_index_physical_stats
(DB_ID(),OBJECT_ID(N'Production.TransactionHistory'), NULL , NULL, NULL);
GO
--Rebuild only partition 5.
ALTER INDEX IX_TransactionHistory_TransactionDate
ON Production.TransactionHistory
REBUILD Partition = 5;
GO
```

I. Changing the compression setting of an index

The following example rebuilds an index on a nonpartitioned table.

```
ALTER INDEX IX_INDEX1
ON T1
REBUILD
WITH ( DATA_COMPRESSION = PAGE )
GO
```

For additional data compression examples, see Creating Compressed Tables and Indexes.

See Also

CREATE INDEX
CREATE SPATIAL INDEX (Transact-SQL)
CREATE XML INDEX (Transact-SQL)
DROP INDEX (Transact-SQL)
Disable Indexes and Constraints
Indexes on xml Type columns

Performing Index Operations Online
Reorganizing and Rebuilding Indexes
sys.dm_db_index_physical_stats
EVENTDATA

ALTER LOGIN

Changes the properties of a SQL Server login account.

 Transact-SQL Syntax Conventions

Syntax

```
ALTER LOGIN login_name
   {
   <status_option>
   | WITH <set_option> [ ,... ]
   | <cryptographic_credential_option>
   }

<status_option> ::=
      ENABLE | DISABLE

<set_option> ::=
   PASSWORD = 'password' | hashed_password HASHED
   [
    OLD_PASSWORD = 'oldpassword'
    | <password_option> [<password_option> ]
   ]
   | DEFAULT_DATABASE = database
   | DEFAULT_LANGUAGE = language
   | NAME = login_name
   | CHECK_POLICY = { ON | OFF }
   | CHECK_EXPIRATION = { ON | OFF }
   | CREDENTIAL = credential_name
   | NO CREDENTIAL

<password_option> ::=
```

MUST_CHANGE | UNLOCK

<cryptographic_credentials_option> ::=

ADD CREDENTIAL **credential_name**

| DROP CREDENTIAL **credential_name**

Arguments

login_name

Specifies the name of the SQL Server login that is being changed. Domain logins must be enclosed in brackets in the format [domain\user].

ENABLE | DISABLE

Enables or disables this login.

PASSWORD = 'password'

Applies only to SQL Server logins. Specifies the password for the login that is being changed. Passwords are case-sensitive.

PASSWORD = hashed_password

Applies to the HASHED keyword only. Specifies the hashed value of the password for the login that is being created.

HASHED

Applies to SQL Server logins only. Specifies that the password entered after the PASSWORD argument is already hashed. If this option is not selected, the password is hashed before being stored in the database. This option should only be used for login synchronization between two servers. Do not use the HASHED option to routinely change passwords.

OLD_PASSWORD = 'oldpassword'

Applies only to SQL Server logins. The current password of the login to which a new password will be assigned. Passwords are case-sensitive.

MUST_CHANGE

Applies only to SQL Server logins. If this option is included, SQL Server will prompt for an updated password the first time the altered login is used.

DEFAULT_DATABASE = database

Specifies a default database to be assigned to the login.

DEFAULT_LANGUAGE = language

Specifies a default language to be assigned to the login.

NAME = login_name

The new name of the login that is being renamed. If this is a Windows login, the SID of the Windows principal corresponding to the new name must match the SID associated with the

login in SQL Server. The new name of a SQL Server login cannot contain a backslash character (\).

CHECK_EXPIRATION = { ON | OFF }

Applies only to SQL Server logins. Specifies whether password expiration policy should be enforced on this login. The default value is OFF.

CHECK_POLICY = { ON | OFF }

Applies only to SQL Server logins. Specifies that the Windows password policies of the computer on which SQL Server is running should be enforced on this login. The default value is ON.

CREDENTIAL = credential_name

The name of a credential to be mapped to a SQL Server login. The credential must already exist in the server. For more information see EVENTDATA (Transact-SQL). A credential cannot be mapped to the sa login.

NO CREDENTIAL

Removes any existing mapping of the login to a server credential. For more information see Credentials.

UNLOCK

Applies only to SQL Server logins. Specifies that a login that is locked out should be unlocked.

ADD CREDENTIAL

Adds an Extensible Key Management (EKM) provider credential to the login. For more information, see Understanding Extensible Key Management (EKM).

DROP CREDENTIAL

Removes an Extensible Key Management (EKM) provider credential to the login. For more information see Understanding Extensible Key Management (EKM).

Remarks

When CHECK_POLICY is set to ON, the HASHED argument cannot be used.

When CHECK_POLICY is changed to ON, the following behavior occurs:

- CHECK_EXPIRATION is also set to ON, unless it is explicitly set to OFF.
- The password history is initialized with the value of the current password hash.

When CHECK_POLICY is changed to OFF, the following behavior occurs:

- CHECK_EXPIRATION is also set to OFF.
- The password history is cleared.
- The value of lockout_time is reset.

If MUST_CHANGE is specified, CHECK_EXPIRATION and CHECK_POLICY must be set to ON. Otherwise, the statement will fail.

If CHECK_POLICY is set to OFF, CHECK_EXPIRATION cannot be set to ON. An ALTER LOGIN statement that has this combination of options will fail.

You cannot use ALTER_LOGIN with the DISABLE argument to deny access to a Windows group. For example, ALTER_LOGIN [*domain**group*] DISABLE will return the following error message:

"Msg 15151, Level 16, State 1, Line 1

"Cannot alter the login '*Domain**Group*', because it does not exist or you do not have permission."

This is by design.

Permissions

Requires ALTER ANY LOGIN permission.

If the CREDENTIAL option is used, also requires ALTER ANY CREDENTIAL permission.

If the login that is being changed is a member of the **sysadmin** fixed server role or a grantee of CONTROL SERVER permission, also requires CONTROL SERVER permission when making the following changes:

- Resetting the password without supplying the old password.
- Enabling MUST_CHANGE, CHECK_POLICY, or CHECK_EXPIRATION.
- Changing the login name.
- Enabling or disabling the login.
- Mapping the login to a different credential.

A principal can change the password, default language, and default database for its own login.

Examples

A. Enabling a disabled login

The following example enables the login `Mary5`.

```
ALTER LOGIN Mary5 ENABLE;
```

B. Changing the password of a login

The following example changes the password of login `Mary5` to a strong password.

```
ALTER LOGIN Mary5 WITH PASSWORD = '<enterStrongPasswordHere>';
```

C. Changing the name of a login

The following example changes the name of login `Mary5` to `John2`.

```
ALTER LOGIN Mary5 WITH NAME = John2;
```

D. Mapping a login to a credential

The following example maps the login `John2` to the credential `Custodian04`.

```
ALTER LOGIN John2 WITH CREDENTIAL = Custodian04;
```

E. Mapping a login to an Extensible Key Management credential

The following example maps the login `Mary5` to the EKM credential `EKMProvider1`.

```
ALTER LOGIN Mary5
ADD CREDENTIAL EKMProvider1;
GO
```

F. Unlocking a login

To unlock a SQL Server login, execute the following statement, replacing **** with the desired account password.

```
ALTER LOGIN [Mary5] WITH PASSWORD = '****' UNLOCK ;
GO
```

To unlock a login without changing the password, turn the check policy off and then on again.

```
ALTER LOGIN [Mary5] WITH CHECK_POLICY = OFF;
ALTER LOGIN [Mary5] WITH CHECK_POLICY = ON;
GO
```

G. Changing the password of a login using HASHED

The following example changes the password of the `TestUser` login to an already hashed value.

```
ALTER LOGIN TestUser WITH
PASSWORD = 0x01000CF35567C60BFB41EBDE4CF700A985A13D773D6B45B90900 HASHED ;
GO
```

See Also

Credentials
CREATE LOGIN (Transact-SQL)
DROP LOGIN (Transact-SQL)
CREATE CREDENTIAL (Transact-SQL)
EVENTDATA (Transact-SQL)
Understanding Extensible Key Management (EKM)

ALTER MASTER KEY

Changes the properties of a database master key.

 Transact-SQL Syntax Conventions

Syntax

ALTER MASTER KEY <alter_option>

<alter_option> ::=

<regenerate_option> | <encryption_option>

<regenerate_option> ::=

[FORCE] REGENERATE WITH ENCRYPTION BY PASSWORD = **'password'**

<encryption_option> ::=

ADD ENCRYPTION BY { SERVICE MASTER KEY | PASSWORD = **'password'** }

|

DROP ENCRYPTION BY { SERVICE MASTER KEY | PASSWORD = **'password'** }

Arguments

PASSWORD = 'password'

Specifies a password with which to encrypt or decrypt the database master key. password must meet the Windows password policy requirements of the computer that is running the instance of SQL Server.

Remarks

The REGENERATE option re-creates the database master key and all the keys it protects. The keys are first decrypted with the old master key, and then encrypted with the new master key. This resource-intensive operation should be scheduled during a period of low demand, unless the master key has been compromised.

SQL Server 2012 uses the AES encryption algorithm to protect the service master key (SMK) and the database master key (DMK). AES is a newer encryption algorithm than 3DES used in earlier versions. After upgrading an instance of the Database Engine to SQL Server 2012 the SMK and DMK should be regenerated in order to upgrade the master keys to AES. For more information about regenerating the SMK, see ALTER SERVICE MASTER KEY (Transact-SQL).

When the FORCE option is used, key regeneration will continue even if the master key is unavailable or the server cannot decrypt all the encrypted private keys. If the master key cannot be opened, use the RESTORE MASTER KEY statement to restore the master key from a backup. Use the FORCE option only if the master key is irretrievable or if decryption fails. Information that is encrypted only by an irretrievable key will be lost.

The DROP ENCRYPTION BY SERVICE MASTER KEY option removes the encryption of the database master key by the service master key.

ADD ENCRYPTION BY SERVICE MASTER KEY causes a copy of the master key to be encrypted using the service master key and stored in both the current database and in master.

Permissions

Requires CONTROL permission on the database. If the database master key has been encrypted with a password, knowledge of that password is also required.

Examples

The following example creates a new database master key for `AdventureWorks` and reencrypts the keys below it in the encryption hierarchy.

```
USE AdventureWorks2012;
ALTER MASTER KEY REGENERATE WITH ENCRYPTION BY PASSWORD =
'dsjdkflJ435907NnmM#sX003';
GO
```

See Also

Detaching and Attaching Databases
OPEN MASTER KEY (Transact-SQL)
CLOSE MASTER KEY (Transact-SQL)
BACKUP MASTER KEY (Transact-SQL)
RESTORE MASTER KEY (Transact-SQL)
DROP MASTER KEY (Transact-SQL)
Encryption Hierarchy
CREATE DATABASE (Transact-SQL)
Detaching and Attaching a Database

ALTER MESSAGE TYPE

Changes the properties of a message type.

 Transact-SQL Syntax Conventions

Syntax

```
ALTER MESSAGE TYPE message_type_name
  VALIDATION =
  { NONE
   | EMPTY
   | WELL_FORMED_XML
   | VALID_XML WITH SCHEMA COLLECTION schema_collection_name }
[;]
```

Arguments

message_type_name

The name of the message type to change. Server, database, and schema names cannot be specified.

VALIDATION

Specifies how Service Broker validates the message body for messages of this type.

NONE

No validation is performed. The message body might contain any data, or might be NULL.

EMPTY

The message body must be NULL.

WELL_FORMED_XML

The message body must contain well-formed XML.

VALID_XML_WITH_SCHEMA = schema_collection_name

The message body must contain XML that complies with a schema in the specified schema collection. The schema_collection_name must be the name of an existing XML schema collection.

Remarks

Changing the validation of a message type does not affect messages that have already been delivered to a queue.

To change the AUTHORIZATION for a message type, use the ALTER AUTHORIZATION statement.

Permissions

Permission for altering a message type defaults to the owner of the message type, members of the **db_ddladmin** or **db_owner** fixed database roles, and members of the **sysadmin** fixed server role.

When the ALTER MESSAGE TYPE statement specifies a schema collection, the user executing the statement must have REFERENCES permission on the schema collection specified.

Examples

The following example changes the message type `//Adventure-Works.com/Expenses/SubmitExpense` to require that the message body contain a well-formed XML document.

```
ALTER MESSAGE TYPE
    [//Adventure-Works.com/Expenses/SubmitExpense]
    VALIDATION = WELL_FORMED_XML ;
```

See Also

EVENTDATA (Transact-SQL)

CREATE MESSAGE TYPE

DROP MESSAGE TYPE

EVENTDATA (Transact-SQL)

ALTER PARTITION FUNCTION

Alters a partition function by splitting or merging its boundary values. By executing ALTER PARTITION FUNCTION, one partition of any table or index that uses the partition function can be split into two partitions, or two partitions can be merged into one less partition.

Caution
More than one table or index can use the same partition function. ALTER PARTITION FUNCTION affects all of them in a single transaction.

Transact-SQL Syntax Conventions

Syntax

```
ALTER PARTITION FUNCTION partition_function_name()
{
   SPLIT RANGE ( boundary_value )
  | MERGE RANGE ( boundary_value )
} [ ; ]
```

Arguments

partition_function_name

Is the name of the partition function to be modified.

SPLIT RANGE (boundary_value)

Adds one partition to the partition function. boundary_value determines the range of the new partition, and must differ from the existing boundary ranges of the partition function. Based on boundary_value, the Database Engine splits one of the existing ranges into two. Of these two, the one where the new boundary_value resides is considered the new partition.

A filegroup must exist online and be marked by the partition scheme that uses the partition function as NEXT USED to hold the new partition. Filegroups are allocated to partitions in a CREATE PARTITION SCHEME statement. If a CREATE PARTITION SCHEME statement allocates more filegroups than necessary (fewer partitions are created in the CREATE PARTITION FUNCTION statement than filegroups to hold them), then there are unassigned filegroups, and one of them is marked NEXT USED by the partition scheme. This filegroup will hold the new partition. If there are no filegroups marked NEXT USED by the partition scheme, you must use ALTER PARTITION SCHEME to either add a filegroup, or designate an existing one, to hold the new partition. A filegroup that already holds partitions can be designated to hold additional partitions. Because a partition function can participate in more than one partition scheme, all the partition schemes that use the partition function to which you are adding partitions must have a NEXT USED filegroup. Otherwise, ALTER PARTITION FUNCTION fails with an error that displays the partition scheme or schemes that lack a NEXT USED filegroup.

If you create all the partitions in the same filegroup, that filegroup is initially assigned to be

the NEXT USED filegroup automatically. However, after a split operation is performed, there is no longer a designated NEXT USED filegroup. You must explicitly assign the filegroup to be the NEXT USED filegroup by using ALTER PARITION SCHEME or a subsequent split operation will fail.

MERGE [RANGE (boundary_value)]

Drops a partition and merges any values that exist in the partition into one of the remaining partitions. RANGE (boundary_value) must be an existing boundary value, into which the values from the dropped partition are merged. The filegroup that originally held boundary_value is removed from the partition scheme unless it is used by a remaining partition, or is marked with the NEXT USED property. The merged partition resides in the filegroup that originally did not hold boundary_value. boundary_value is a constant expression that can reference variables (including user-defined type variables) or functions (including user-defined functions). It cannot reference a Transact-SQL expression. boundary_value must either match or be implicitly convertible to the data type of its corresponding partitioning column, and cannot be truncated during implicit conversion in a way that the size and scale of the value does not match that of its corresponding input_parameter_type.

Best Practices

Always keep empty partitions at both ends of the partition range to guarantee that the partition split (before loading new data) and partition merge (after unloading old data) do not incur any data movement. Avoid splitting or merging populated partitions. This can be extremely inefficient, as this may cause as much as four times more log generation, and may also cause severe locking.

Limitations and Restrictions

ALTER PARTITION FUNCTION repartitions any tables and indexes that use the function in a single atomic operation. However, this operation occurs offline, and depending on the extent of repartitioning, may be resource-intensive.

ALTER PARTITION FUNCTION can only be used for splitting one partition into two, or merging two partitions into one. To change the way a table is otherwise partitioned (for example, from 10 partitions to 5 partitions), you can exercise any of the following options. Depending on the configuration of your system, these options can vary in resource consumption:

- Create a new partitioned table with the desired partition function, and then insert the data from the old table into the new table by using an INSERT INTO...SELECT FROM statement.
- Create a partitioned clustered index on a heap.

  Note
  Dropping a partitioned clustered index results in a partitioned heap.
- Drop and rebuild an existing partitioned index by using the Transact-SQL CREATE INDEX statement with the DROP EXISTING = ON clause.
- Perform a sequence of ALTER PARTITION FUNCTION statements.

All filegroups that are affected by ALTER PARITITION FUNCTION must be online.

ALTER PARTITION FUNCTION fails when there is a disabled clustered index on any tables that use the partition function.

SQL Server does not provide replication support for modifying a partition function. Changes to a partition function in the publication database must be manually applied in the subscription database.

Permissions

Any one of the following permissions can be used to execute ALTER PARTITION FUNCTION:

- ALTER ANY DATASPACE permission. This permission defaults to members of the **sysadmin** fixed server role and the **db_owner** and **db_ddladmin** fixed database roles.
- CONTROL or ALTER permission on the database in which the partition function was created.
- CONTROL SERVER or ALTER ANY DATABASE permission on the server of the database in which the partition function was created.

Examples

A. Splitting a partition of a partitioned table or index into two partitions

The following example creates a partition function to partition a table or index into four partitions. `ALTER PARTITION FUNCTION` splits one of the partitions into two to create a total of five partitions.

```
IF EXISTS (SELECT * FROM sys.partition_functions
    WHERE name = 'myRangePF1')
DROP PARTITION FUNCTION myRangePF1;
GO
CREATE PARTITION FUNCTION myRangePF1 (int)
AS RANGE LEFT FOR VALUES ( 1, 100, 1000 );
GO
--Split the partition between boundary_values 100 and 1000
--to create two partitions between boundary_values 100 and 500
--and between boundary_values 500 and 1000.
ALTER PARTITION FUNCTION myRangePF1 ()
SPLIT RANGE (500);
```

B. Merging two partitions of a partitioned table into one partition

The following example creates the same partition function as above, and then merges two of the partitions into one partition, for a total of three partitions.

```
IF EXISTS (SELECT * FROM sys.partition_functions
```

```
    WHERE name = 'myRangePF1')
DROP PARTITION FUNCTION myRangePF1;
GO
CREATE PARTITION FUNCTION myRangePF1 (int)
AS RANGE LEFT FOR VALUES ( 1, 100, 1000 );
GO
--Merge the partitions between boundary_values 1 and 100
--and between boundary_values 100 and 1000 to create one partition
--between boundary_values 1 and 1000.
ALTER PARTITION FUNCTION myRangePF1 ()
MERGE RANGE (100);
```

See Also

Partitioned Tables and Indexes
sys.index_columns (Transact-SQL)
DROP PARTITION FUNCTION
CREATE PARTITION SCHEME
ALTER PARTITION SCHEME
DROP PARTITION SCHEME
CREATE INDEX
ALTER INDEX
CREATE TABLE
sys.partition_functions
sys.partition_parameters
sys.partition_range_values
sys.partitions
sys.tables
sys.indexes
sys.index_columns

ALTER PARTITION SCHEME

Adds a filegroup to a partition scheme or alters the designation of the NEXT USED filegroup for the partition scheme.

 Transact-SQL Syntax Conventions

Syntax

ALTER PARTITION SCHEME **partition_scheme_name**
NEXT USED [**filegroup_name**] [;]

Arguments

partition_scheme_name

Is the name of the partition scheme to be altered.

filegroup_name

Specifies the filegroup to be marked by the partition scheme as NEXT USED. This means the filegroup will accept a new partition that is created by using an ALTER PARTITION FUNCTION statement.

In a partition scheme, only one filegroup can be designated NEXT USED. A filegroup that is not empty can be specified. If filegroup_name is specified and there currently is no filegroup marked NEXT USED, filegroup_name is marked NEXT USED. If filegroup_name is specified, and a filegroup with the NEXT USED property already exists, the NEXT USED property transfers from the existing filegroup to filegroup_name.

If filegroup_name is not specified and a filegroup with the NEXT USED property already exists, that filegroup loses its NEXT USED state so that there are no NEXT USED filegroups in partition_scheme_name.

If filegroup_name is not specified, and there are no filegroups marked NEXT USED, ALTER PARTITION SCHEME returns a warning.

Remarks

Any filegroup affected by ALTER PARTITION SCHEME must be online.

Permissions

Tthe following permissions can be used to execute ALTER PARTITION SCHEME:

- ALTER ANY DATASPACE permission. This permission defaults to members of the **sysadmin** fixed server role and the **db_owner** and **db_ddladmin** fixed database roles.
- CONTROL or ALTER permission on the database in which the partition scheme was created.
- CONTROL SERVER or ALTER ANY DATABASE permission on the server of the database in which the partition scheme was created.

Examples

The following example assumes the partition scheme `MyRangePS1` and the filegroup `test5fg` exist in the current database.

```
ALTER PARTITION SCHEME MyRangePS1
NEXT USED test5fg;
```

Filegroup `test5fg` will receive any additional partition of a partitioned table or index as a result of an ALTER PARTITION FUNCTION statement.

See Also

sys.index_columns (Transact-SQL)
DROP PARTITION SCHEME
CREATE PARTITION FUNCTION
ALTER PARTITION FUNCTION
DROP PARTITION FUNCTION
CREATE TABLE
CREATE INDEX
EVENTDATA
sys.partition_schemes
sys.data_spaces
sys.destination_data_spaces
sys.partitions
sys.tables
sys.indexes
sys.index_columns

ALTER PROCEDURE

Modifies a previously created procedure that was created by executing the CREATE PROCEDURE statement in SQL Server 2012.

 Transact-SQL Syntax Conventions (Transact-SQL)

Syntax

```
--Transact-SQL Stored Procedure Syntax
ALTER { PROC | PROCEDURE } [schema_name.] procedure_name [ ;number ]
    [ { @parameter [ type_schema_name. ] data_type }
        [ VARYING ] [ = default ] [ OUT | OUTPUT ] [READONLY]
    ] [ ,...n ]
[ WITH <procedure_option> [ ,...n ] ]
[ FOR REPLICATION ]
AS { [ BEGIN ] sql_statement [;] [ ...n ] [ END ] }
[;]
```

<procedure_option> ::=

[ENCRYPTION]

[RECOMPILE]

[**EXECUTE AS Clause**]

--CLR Stored Procedure Syntax

ALTER { PROC | PROCEDURE } [**schema_name.**] **procedure_name** [**; number**]

[{ **@parameter** [**type_schema_name.**] **data_type** }

[**= default**] [OUT | OUTPUT] [READONLY]

] [**,...n**]

[WITH **EXECUTE AS Clause**]

AS { EXTERNAL NAME **assembly_name.class_name.method_name** }

[;]

Arguments

schema_name

The name of the schema to which the procedure belongs.

procedure_name

The name of the procedure to change. Procedure names must comply with the rules for identifiers.

; number

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

Note

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

@ parameter

A parameter in the procedure. Up to 2,100 parameters can be specified.

[type_schema_name.] data_type

Is the data type of the parameter and the schema it belongs to.

For information about data type restrictions, see CREATE PROCEDURE (Transact-SQL).

VARYING

Specifies the result set supported as an output parameter. This parameter is constructed dynamically by the stored procedure and its contents can vary. Applies only to cursor

parameters. This option is not valid for CLR procedures.

default

Is a default value for the parameter.

OUT | OUTPUT

Indicates that the parameter is a return parameter.

READONLY

Indicates that the parameter cannot 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 does not cache a plan for this procedure and the procedure is recompiled at run time.

ENCRYPTION

Indicates that the Database Engine will convert the original text of the ALTER PROCEDURE statement to an obfuscated format. The output of the obfuscation is not directly visible in any of the catalog views in SQL Server. Users that have no access to system tables or database files cannot retrieve the obfuscated text. However, the text will be available to privileged users that can either access system tables over the DAC port or directly access database files. Also, users that can attach a debugger to the server process can retrieve the original procedure from memory at runtime. For more information about accessing system metadata, see Metadata Visibility Configuration.

Procedures created with this option cannot be published as part of SQL Server replication.

This option cannot be specified for common language runtime (CLR) stored procedures.

Note

During an upgrade, the Database Engine uses the obfuscated comments stored in **sys.sql_modules** to re-create procedures.

EXECUTE AS

Specifies the security context under which to execute the stored procedure after it is accessed.

For more information, see EXECUTE AS Clause (Transact-SQL).

FOR REPLICATION

Specifies that stored procedures that are created for replication cannot be executed on the Subscriber. A stored procedure created with the FOR REPLICATION option is used as a stored procedure filter and only executed during replication. Parameters cannot be declared if FOR REPLICATION is specified. This option is not valid for CLR procedures. The RECOMPILE option is ignored for procedures created with FOR REPLICATION.

Note

This option is not available in a contained database.

{ [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 more information, see the Best Practices, General Remarks, and Limitations and Restrictions sections in CREATE PROCEDURE (Transact-SQL).

EXTERNAL NAME assembly_name.class_name.method_name

Specifies the method of a .NET Framework assembly for a CLR stored 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 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 cannot execute CLR code. You can create, modify, and drop database objects that reference common language runtime modules; however, you cannot 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.

General Remarks

Transact-SQL stored procedures cannot be modified to be CLR stored procedures and vice versa.

ALTER PROCEDURE does not change permissions and does not affect any dependent stored procedures or triggers. However, the current session settings for QUOTED_IDENTIFIER and ANSI_NULLS are included in the stored procedure when it is modified. If the settings are different from those in effect when stored procedure was originally created, the behavior of the stored procedure may change.

If a previous procedure definition was created using WITH ENCRYPTION or WITH RECOMPILE, these options are enabled only if they are included in ALTER PROCEDURE.

For more information about stored procedures, see CREATE PROCEDURE (Transact-SQL).

Security

Permissions

Requires **ALTER** permission on the procedure or requires membership in the **db_ddladmin** fixed database role.

Examples

The following example creates the `uspVendorAllInfo` stored procedure. This procedure returns the names of all the vendors that supply Adventure Works Cycles, the products they supply,

their credit ratings, and their availability. After this procedure is created, it is then modified to return a different result set.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID ( 'Purchasing.uspVendorAllInfo', 'P' ) IS NOT NULL
    DROP PROCEDURE Purchasing.uspVendorAllInfo;
GO
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
```

The following example alters the `uspVendorAllInfo` stored procedure. It removes the EXECUTE AS CALLER clause and modifies the body of the procedure to return only those vendors that supply the specified product. The `LEFT` and `CASE` functions customize the appearance of the result set.

```
USE AdventureWorks2012;
GO
ALTER PROCEDURE Purchasing.uspVendorAllInfo
    @Product varchar(25)
AS
    SET NOCOUNT ON;
    SELECT LEFT(v.Name, 25) AS Vendor, LEFT(p.Name, 25) AS 'Product name',
    'Rating' = CASE v.CreditRating
        WHEN 1 THEN 'Superior'
        WHEN 2 THEN 'Excellent'
```

```
        WHEN 3 THEN 'Above average'
        WHEN 4 THEN 'Average'
        WHEN 5 THEN 'Below average'
        ELSE 'No rating'
        END
    , Availability = CASE v.ActiveFlag
        WHEN 1 THEN 'Yes'
        ELSE 'No'
        END
    FROM Purchasing.Vendor AS v
    INNER JOIN Purchasing.ProductVendor AS pv
      ON v.BusinessEntityID = pv.BusinessEntityID
    INNER JOIN Production.Product AS p
      ON pv.ProductID = p.ProductID
    WHERE p.Name LIKE @Product
    ORDER BY v.Name ASC;
GO
```

Here is the result set.

```
Vendor                Product name  Rating    Availability
--------------------- ------------- -------   ------------
Proseware, Inc.       LL Crankarm   Average   No
Vision Cycles, Inc.   LL Crankarm   Superior  Yes

(2 row(s) affected)
```

See Also

CREATE PROCEDURE (Transact-SQL)
DROP PROCEDURE
EXECUTE (Transact-SQL)
EXECUTE AS (Transact-SQL)
EVENTDATA (Transact-SQL)
Stored Procedures (Database Engine)
sys.procedures (Transact-SQL)

ALTER QUEUE

Changes the properties of a queue.

 Transact-SQL Syntax Conventions

Syntax

```
ALTER QUEUE <object> WITH
  [ STATUS = { ON | OFF } [ , ] ]
  [ RETENTION = { ON | OFF } [ , ] ]
  [ ACTIVATION (
    { [ STATUS = { ON | OFF } [ , ] ]
      [ PROCEDURE_NAME = <procedure> [ , ] ]
      [ MAX_QUEUE_READERS = max_readers [ , ] ]
      [ EXECUTE AS { SELF | 'user_name' | OWNER } ]
    | DROP }
      ) [ , ]]
      [ POISON_MESSAGE_HANDLING (
       STATUS = { ON | OFF } )
      ]
 [ ; ]

<object> ::=
{
  [ database_name. [ schema_name ] . | schema_name. ]
    queue_name
}

<procedure> ::=
{
  [ database_name. [ schema_name ] . | schema_name. ]
    stored_procedure_name
}
```

Arguments

database_name (object)

Is the name of the database that contains the queue to be changed. When no database_name is provided, this defaults to the current database.

schema_name (object)

Is the name of the schema to which the new queue belongs. When no schema_name is provided, this defaults to the default schema for the current user.

queue_name

Is the name of the queue to be changed.

STATUS (Queue)

Specifies whether the queue is available (ON) or unavailable (OFF). When the queue is unavailable, no messages can be added to the queue or removed from the queue.

RETENTION

Specifies the retention setting for the queue. If RETENTION = ON, all messages sent or received on conversations using this queue are retained in the queue until the conversations have ended. This allows you to retain messages for auditing purposes, or to perform compensating transactions if an error occurs

Note

Setting RETENTION = ON can reduce performance. This setting should only be used if required to meet the service level agreement for the application.

ACTIVATION

Specifies information about the stored procedure that is activated to process messages that arrive in this queue.

STATUS (Activation)

Specifies whether or not the queue activates the stored procedure. When STATUS = ON, the queue starts the stored procedure specified with PROCEDURE_NAME when the number of procedures currently running is less than MAX_QUEUE_READERS and when messages arrive on the queue faster than the stored procedures receive messages. When STATUS = OFF, the queue does not activate the stored procedure.

PROCEDURE_NAME = <procedure>

Specifies the name of the stored procedure to activate when the queue contains messages to be processed. This value must be a SQL Server identifier.

database_name (procedure)

Is the name of the database that contains the stored procedure.

schema_name (procedure)

Is the name of the schema that owns the stored procedure.

stored_procedure_name

Is the name of the stored procedure.

MAX_QUEUE_READERS = max_reader

Specifies the maximum number of instances of the activation stored procedure that the queue starts simultaneously. The value of max_readers must be a number between 0 and 32767.

EXECUTE AS

Specifies the SQL Server database user account under which the activation stored procedure runs. SQL Server must be able to check the permissions for this user at the time that the queue activates the stored procedure. For Windows domain user, the SQL Server must be connected to the domain and able to validate the permissions of the specified user when the procedure is activated or activation fails. For a SQL Server user, the server can always check permissions.

SELF

Specifies that the stored procedure executes as the current user. (The database principal executing this ALTER QUEUE statement.)

'user_name'

Is the name of the user that the stored procedure executes as. user_name must be a valid SQL Server user specified as a SQL Server identifier. The current user must have IMPERSONATE permission for the user_name specified.

OWNER

Specifies that the stored procedure executes as the owner of the queue.

DROP

Deletes all of the activation information associated with the queue.

POISON_MESSAGE_HANDLING

Specifies whether poison message handling is enabled. The default is ON.

A queue that has poison message handling set to OFF will not be disabled after five consecutive transaction rollbacks. This allows for a custom poison message handing system to be defined by the application.

Remarks

When a queue with a specified activation stored procedure contains messages, changing the activation status from OFF to ON immediately activates the activation stored procedure. Altering the activation status from ON to OFF stops the broker from activating instances of the stored procedure, but does not stop instances of the stored procedure that are currently running.

Altering a queue to add an activation stored procedure does not change the activation status of the queue. Changing the activation stored procedure for the queue does not affect instances of the activation stored procedure that are currently running.

Service Broker checks the maximum number of queue readers for a queue as part of the activation process. Therefore, altering a queue to increase the maximum number of queue readers allows Service Broker to immediately start more instances of the activation stored procedure. Altering a queue to decrease the maximum number of queue readers does not affect instances of the activation stored procedure currently running. However, Service Broker does not start a new instance of the stored procedure until the number of instances for the activation stored procedure falls below the configured maximum number.

When a queue is unavailable, Service Broker holds messages for services that use the queue in the transmission queue for the database. The sys.transmission_queue catalog view provides a view of the transmission queue.

If a RECEIVE statement or a GET CONVERSATION GROUP statement specifies an unavailable queue, that statement fails with a Transact-SQL error.

Permissions

Permission for altering a queue defaults to the owner of the queue, members of the db_ddladmin or db_owner fixed database roles, and members of the sysadmin fixed server role.

Examples

A. Making a queue unavailable

The following example makes the `ExpenseQueue` queue unavailable to receive messages.

```
ALTER QUEUE ExpenseQueue WITH STATUS = OFF ;
```

B. Changing the activation stored procedure

The following example changes the stored procedure that the queue starts. The stored procedure executes as the user who ran the `ALTER QUEUE` statement.

```
ALTER QUEUE ExpenseQueue
    WITH ACTIVATION (
        PROCEDURE_NAME = new_stored_proc,
        EXECUTE AS SELF) ;
```

C. Changing the number of queue readers

The following example sets to `7` the maximum number of stored procedure instances that Service Broker starts for this queue.

```
ALTER QUEUE ExpenseQueue WITH ACTIVATION (MAX_QUEUE_READERS = 7) ;
```

D. Changing the activation stored procedure and the EXECUTE AS account

The following example changes the stored procedure that Service Broker starts. The stored procedure executes as the user `SecurityAccount`.

```
ALTER QUEUE ExpenseQueue
    WITH ACTIVATION (
```

```
    PROCEDURE_NAME = AdventureWorks2012.dbo.new_stored_proc ,
    EXECUTE AS 'SecurityAccount') ;
```

E. Setting the queue to retain messages

The following example sets the queue to retain messages. The queue retains all messages sent to or from services that use this queue until the conversation that contains the message ends.

```
ALTER QUEUE ExpenseQueue WITH RETENTION = ON ;
```

F. Removing activation from a queue

The following example removes all activation information from the queue.

```
ALTER QUEUE ExpenseQueue WITH ACTIVATION (DROP) ;
```

See Also

CREATE QUEUE
DROP QUEUE
EVENTDATA

ALTER REMOTE SERVICE BINDING

Changes the user associated with a remote service binding, or changes the anonymous authentication setting for the binding.

 Transact-SQL Syntax Conventions

Syntax

ALTER REMOTE SERVICE BINDING **binding_name**
  WITH [USER = <**user_name**>] [, ANONYMOUS = { ON | OFF }]
[;]

Arguments

binding_name

The name of the remote service binding to change. Server, database, and schema names cannot be specified.

WITH USER = <user_name>

Specifies the database user that holds the certificate associated with the remote service for this binding. The public key from this certificate is used for encryption and authentication of messages exchanged with the remote service.

ANONYMOUS

Specifies whether anonymous authentication is used when communicating with the remote service. If ANONYMOUS = ON, anonymous authentication is used and the credentials of the

local user are not transferred to the remote service. If ANONYMOUS = OFF, user credentials are transferred. If this clause is not specified, the default is OFF.

Remarks

The public key in the certificate associated with user_name is used to authenticate messages sent to the remote service and to encrypt a session key that is then used to encrypt the conversation. The certificate for user_name must correspond to the certificate for a login in the database that hosts the remote service.

Permissions

Permission for altering a remote service binding defaults to the owner of the remote service binding, members of the **db_owner** fixed database role, and members of the **sysadmin** fixed server role.

The user that executes the ALTER REMOTE SERVICE BINDING statement must have impersonate permission for the user specified in the statement.

To alter the AUTHORIZATION for a remote service binding, use the ALTER AUTHORIZATION statement.

Examples

The following example changes the remote service binding `APBinding` to encrypt messages by using the certificates from the account `SecurityAccount`.

```
ALTER REMOTE SERVICE BINDING APBinding
    WITH USER = SecurityAccount ;
```

See Also

EVENTDATA (Transact-SQL)

DROP REMOTE SERVICE BINDING

EVENTDATA

ALTER RESOURCE GOVERNOR

This command is used to perform the following actions:

- Apply the configuration changes specified when the CREATE|ALTER|DROP WORKLOAD GROUP or CREATE|ALTER|DROP RESOURCE POOL statements are issued.
- Enable or disable Resource Governor.
- Configure classification for incoming requests.
- Reset workload group and resource pool statistics.

 Transact-SQL Syntax Conventions

Syntax

```
ALTER RESOURCE GOVERNOR
   { DISABLE | RECONFIGURE }
|
   WITH ( CLASSIFIER_FUNCTION = { schema_name.function_name | NULL } )
|
   RESET STATISTICS
[ ; ]
```

Arguments

Term	Definition
DISABLE | RECONFIGURE	DISABLE disables Resource Governor. Disabling Resource Governor has the following results: • The classifier function is not executed. • All new connections are automatically classified into the default group. • System-initiated requests are classified into the internal workload group. • All existing workload group and resource pool settings are reset to their default values. In this case, no events are fired when limits are reached. • Normal system monitoring is not affected. • Configuration changes can be made, but the changes do not take effect until Resource Governor is enabled. • Upon restarting SQL Server, the Resource Governor will not load its configuration, but instead will have only the default and internal groups and pools. When the Resource Governor is not enabled, RECONFIGURE enables the Resource Governor. Enabling Resource Governor has the following results: • The classifier function is executed for new connections so that their workload

	can be assigned to workload groups. • The resource limits that are specified in the Resource Governor configuration are honored and enforced. • Requests that existed before enabling Resource Governor are affected by any configuration changes that were made when Resource Governor was disabled. When Resource Governor is running, RECONFIGURE applies any configuration changes requested when the CREATE|ALTER|DROP WORKLOAD GROUP or CREATE|ALTER|DROP RESOURCE POOL statements are executed. **Important** ALTER RESOURCE GOVERNOR RECONFIGURE must be issued in order for any configuration changes to take effect.
CLASSIFIER_FUNCTION = { schema_name.function_name | NULL }	Registers the classification function specified by *schema_name.function_name*. This function classifies every new session and assigns the session requests and queries to a workload group. When NULL is used, new sessions are automatically assigned to the default workload group.
RESET STATISTICS	Resets statistics on all workload groups and resource pools. For more information, see sys.dm_resource_governor_workload_groups (Transact-SQL) and sys.dm_resource_governor_resource_pools (Transact-SQL).

Remarks

ALTER RESOURCE GOVERNOR DISABLE, ALTER RESOURCE GOVERNOR RECONFIGURE, and ALTER RESOURCE GOVERNOR RESET STATISTICS cannot be used inside a user transaction.

The RECONFIGURE parameter is part of the Resource Governor syntax and should not be confused with RECONFIGURE, which is a separate DDL statement.

We recommend being familiar with Resource Governor states before you execute DDL statements. For more information, see Resource Governor.

Permissions

Requires CONTROL SERVER permission.

Examples

A. Starting the Resource Governor

When SQL Server is first installed Resource Governor is disabled. The following example starts Resource Governor. After the statement executes, Resource Governor is running and can use the predefined workload groups and resource pools.

```
ALTER RESOURCE GOVERNOR RECONFIGURE;
```

B. Assigning new sessions to the default group

The following example assigns all new sessions to the default workload group by removing any existing classifier function from the Resource Governor configuration. When no function is designated as a classifier function, all new sessions are assigned to the default workload group. This change applies to new sessions only. Existing sessions are not affected.

```
ALTER RESOURCE GOVERNOR WITH (CLASSIFIER_FUNCTION = NULL);
GO
ALTER RESOURCE GOVERNOR RECONFIGURE;
```

C. Creating and registering a classifier function

The following example creates a classifier function named `dbo.rgclassifier_v1`. The function classifies every new session based on either the user name or application name and assigns the session requests and queries to a specific workload group. Sessions that do not map to the specified user or application names are assigned to the default workload group. The classifier function is then registered and the configuration change is applied.

```
-- Store the classifier function in the master database.
USE master;
GO
SET ANSI_NULLS ON;
GO
SET QUOTED_IDENTIFIER ON;
GO
CREATE FUNCTION dbo.rgclassifier_v1() RETURNS sysname
WITH SCHEMABINDING
AS
BEGIN
```

```
-- Declare the variable to hold the value returned in sysname.
    DECLARE @grp_name AS sysname
-- If the user login is 'sa', map the connection to the groupAdmin
-- workload group.
    IF (SUSER_NAME() = 'sa')
        SET @grp_name = 'groupAdmin'
-- Use application information to map the connection to the groupAdhoc
-- workload group.
    ELSE IF (APP_NAME() LIKE '%MANAGEMENT STUDIO%')
        OR (APP_NAME() LIKE '%QUERY ANALYZER%')
            SET @grp_name = 'groupAdhoc'
-- If the application is for reporting, map the connection to
-- the groupReports workload group.
    ELSE IF (APP_NAME() LIKE '%REPORT SERVER%')
        SET @grp_name = 'groupReports'
-- If the connection does not map to any of the previous groups,
-- put the connection into the default workload group.
    ELSE
        SET @grp_name = 'default'
    RETURN @grp_name
END
GO
-- Register the classifier user-defined function and update the
-- the in-memory configuration.
ALTER RESOURCE GOVERNOR WITH (CLASSIFIER_FUNCTION=dbo.rgclassifier_v1);
GO
ALTER RESOURCE GOVERNOR RECONFIGURE;
GO
```

D. Resetting Statistics

The following example resets all workload group and pool statistics.

```
ALTER RESOURCE GOVERNOR RESET STATISTICS;
```

See Also

CREATE RESOURCE POOL (Transact-SQL)

ALTER RESOURCE POOL (Transact-SQL)
DROP RESOURCE POOL (Transact-SQL)
CREATE WORKLOAD GROUP (Transact-SQL)
ALTER WORKLOAD GROUP (Transact-SQL)
DROP WORKLOAD GROUP (Transact-SQL)
Managing SQL Server Workloads with Resource Governor
sys.dm_resource_governor_workload_groups (Transact-SQL)
sys.dm_resource_governor_resource_pools (Transact-SQL)

ALTER RESOURCE POOL

Changes an existing Resource Governor resource pool configuration.

 Transact-SQL Syntax Conventions.The introduction is required.

Syntax

```
ALTER RESOURCE POOL { pool_name | "default" }
[WITH
    ( [ MIN_CPU_PERCENT = value ]
    [ [ , ] MAX_CPU_PERCENT = value ]
    [ [ , ] CAP_CPU_PERCENT = value ]
    [ [ , ] AFFINITY {SCHEDULER = AUTO | (Scheduler_range_spec) | NUMANODE =
(NUMA_node_range_spec)}]
    [ [ , ] MIN_MEMORY_PERCENT = value ]
    [ [ , ] MAX_MEMORY_PERCENT = value ] )
]
[;]

Scheduler_range_spec::=
{SCHED_ID | SCHED_ID TO SCHED_ID}[,...n]
NUMA_node_range_spec::=
{NUMA_node_ID | NUMA_node_ID TO NUMA_node_ID}[,...n]
```

Arguments

{ pool_name | "default" }

Is the name of an existing user-defined resource pool or the default resource pool that is created when SQL Server 2012 is installed.

"default" must be enclosed by quotation marks ("") or brackets ([]) when used with ALTER

RESOURCE POOL to avoid conflict with DEFAULT, which is a system reserved word. For more information, see Database Identifiers.

Note

Predefined workload groups and resource pools all use lowercase names, such as "default". This should be taken into account for servers that use case-sensitive collation. Servers with case-insensitive collation, such as SQL_Latin1_General_CP1_CI_AS, will treat "default" and "Default" as the same.

MIN_CPU_PERCENT = value

Specifies the guaranteed average CPU bandwidth for all requests in the resource pool when there is CPU contention. value is an integer with a default setting of 0. The allowed range for value is from 0 through 100.

MAX_CPU_PERCENT = value

Specifies the maximum average CPU bandwidth that all requests in the resource pool will receive when there is CPU contention. value is an integer with a default setting of 100. The allowed range for value is from 1 through 100.

CAP_CPU_PERCENT = value

Specifies a hard cap on the CPU bandwidth that all requests in the resource pool will receive. Limits the maximum CPU bandwidth level to be the same as the specified value. value is an integer with a default setting of 100. The allowed range for value is from 1 through 100.

AFFINITY {SCHEDULER = AUTO | (Scheduler_range_spec) | NUMANODE = (NUMA_node_range_spec)}

Attach the resource pool to specific schedulers. The default value is AUTO.

MIN_MEMORY_PERCENT = value

Specifies the minimum amount of memory reserved for this resource pool that can not be shared with other resource pools. value is an integer with a default setting of 0. The allowed range for value is from 0 through 100.

MAX_MEMORY_PERCENT = value

Specifies the total server memory that can be used by requests in this resource pool. value is an integer with a default setting of 100. The allowed range for value is from 1 through 100.

Remarks

MAX_CPU_PERCENT and MAX_MEMORY_PERCENT must be greater than or equal to MIN_CPU_PERCENT and MIN_MEMORY_PERCENT, respectively.

CAP_CPU_PERCENT differs from MAX_CPU_PERCENT in that workloads associated with the pool can use CPU capacity above the value of MAX_CPU_PERCENT if it is available, but not above the value of CAP_CPU_PERCENT.

The total CPU percentage for each affinitized component (scheduler(s) or NUMA node(s)) should not exceed 100%.

When you are executing DDL statements, we recommend that you be familiar with Resource Governor states. For more information, see Resource Governor.

Permissions

Requires CONTROL SERVER permission.

Examples

The following example keeps all the default resource pool settings on the `default` pool except for `MAX_CPU_PERCENT`, which is changed to `25`.

```
ALTER RESOURCE POOL "default"
WITH
     ( MAX_CPU_PERCENT = 25)
GO
ALTER RESOURCE GOVERNOR RECONFIGURE
GO
```

In the following example, the `CAP_CPU_PERCENT` sets the hard cap to 80% and `AFFINITY SCHEDULER` is set to an individual value of 8 and a range of 12 to 16.

```
ALTER RESOURCE POOL Pool25
WITH(
     MIN_CPU_PERCENT = 5,
     MAX_CPU_PERCENT = 10,
     CAP_CPU_PERCENT = 80,
     AFFINITY SCHEDULER = (8, 12 TO 16),
     MIN_MEMORY_PERCENT = 5,
     MAX_MEMORY_PERCENT = 15
);

GO
ALTER RESOURCE GOVERNOR RECONFIGURE
GO
```

See Also

Resource Governor
CREATE RESOURCE POOL (Transact-SQL)
DROP RESOURCE POOL (Transact-SQL)
CREATE WORKLOAD GROUP (Transact-SQL)

ALTER WORKLOAD GROUP (Transact-SQL)
DROP WORKLOAD GROUP (Transact-SQL)
ALTER RESOURCE GOVERNOR (Transact-SQL)

ALTER ROLE

Adds members to a database role or changes the name of a user-defined database role.

 Transact-SQL Syntax Conventions

Syntax

```
ALTER ROLE role_name
{
    [ ADD MEMBER database_principal ]
  | [ DROP MEMBER database_principal ]
  | WITH NAME = new_name
}
```

Arguments

role_name

Is the name of the role to be changed.

ADD MEMBER database_principal

Adds the specified database principal to the database role. *database_principal* can be a user or a user-defined database role. *database_principal* cannot be a fixed database role, or a server principal.

DROP MEMBER database_principal

Removes the specified database principal from the database role. *database_principal* can be a user or a user-defined database role. *database_principal* cannot be a fixed database role, a server principal.

WITH NAME = new_name

Specifies the new name of the user-defined role. This name must not already exist in the database. You cannot change the name of fixed database roles.

Remarks

Changing the name of a database role does not change ID number, owner, or permissions of the role.

Database roles are visible in the sys.database_role_members and sys.database_principals catalog views.

Caution

Beginning with SQL Server 2005, the behavior of schemas changed. As a result, code that assumes that schemas are equivalent to database users may no longer return correct results. Old catalog views, including , should not be used in a database in which any of the following DDL statements have ever been used: CREATE SCHEMA, ALTER SCHEMA, DROP SCHEMA, CREATE USER, ALTER USER, DROP USER, CREATE ROLE, ALTER ROLE, DROP ROLE, CREATE APPROLE, ALTER APPROLE, DROP APPROLE, ALTER AUTHORIZATION. In such databases you must instead use the new catalog views. The new catalog views take into account the separation of principals and schemas that was introduced in SQL Server 2005. For more information about catalog views, see .

Permissions

Requires **ALTER ANY ROLE** permission on the database, or **ALTER** permission on the role, or membership in the **db_securityadmin**.

Examples

A. Changing the Name of a Database Role

The following example changes the name of role `buyers` to `purchasing`.

```
USE AdventureWorks2012;
ALTER ROLE buyers WITH NAME = purchasing;
GO
```

B. Adding and Removing Role Members

The following example creates a role named `Sales`, adds and then removes a user named `Barry`.

```
CREATE ROLE Sales;
ALTER ROLE Sales ADD MEMBER Barry;
ALTER ROLE Sales DROP MEMBER Barry;
```

See Also

CREATE ROLE (Transact-SQL)
Principals
DROP ROLE (Transact-SQL)
sp_addrolemember (Transact-SQL)
sys.database_role_members (Transact-SQL)
sys.database_principals (Transact-SQL)

ALTER ROUTE

Modifies route information for an existing route.

Transact-SQL Syntax Conventions

Syntax

```
ALTER ROUTE route_name
WITH
  [ SERVICE_NAME = 'service_name' [ , ] ]
  [ BROKER_INSTANCE = 'broker_instance' [ , ] ]
  [ LIFETIME = route_lifetime [ , ] ]
  [ ADDRESS = 'next_hop_address' [ , ] ]
  [ MIRROR_ADDRESS = 'next_hop_mirror_address' ]
[ ; ]
```

Arguments

route_name

Is the name of the route to change. Server, database, and schema names cannot be specified.

WITH

Introduces the clauses that define the route being altered.

SERVICE_NAME = 'service_name'

Specifies the name of the remote service that this route points to. The service_name must exactly match the name the remote service uses. Service Broker uses a byte-by-byte comparison to match the service_name. In other words, the comparison is case sensitive and does not consider the current collation. A route with a service name of **'SQL/ServiceBroker/BrokerConfiguration'** is a route to a Broker Configuration Notice service. A route to this service might not specify a broker instance.

If the SERVICE_NAME clause is omitted, the service name for the route is unchanged.

BROKER_INSTANCE = 'broker_instance'

Specifies the database that hosts the target service. The broker_instance parameter must be the broker instance identifier for the remote database, which can be obtained by running the following query in the selected database:

```
SELECT service_broker_guid
FROM sys.databases
WHERE database_id = DB_ID()
```

When the BROKER_INSTANCE clause is omitted, the broker instance for the route is unchanged.

Note

This option is not available in a contained database.

LIFETIME = route_lifetime

Specifies the time, in seconds, that SQL Server retains the route in the routing table. At the end of the lifetime, the route expires, and SQL Server no longer considers the route when choosing a route for a new conversation. If this clause is omitted, the lifetime of the route is unchanged.

ADDRESS = 'next_hop_address'

Specifies the network address for this route. The next_hop_address specifies a TCP/IP address in the following format:

TCP:// { *dns_name* | *netbios_name* | *ip_address* } **:** *port_number*

The specified *port_number* must match the port number for the Service Broker endpoint of an instance of SQL Server at the specified computer. This can be obtained by running the following query in the selected database:

```
SELECT tcpe.port
FROM sys.tcp_endpoints AS tcpe
INNER JOIN sys.service_broker_endpoints AS ssbe
   ON ssbe.endpoint_id = tcpe.endpoint_id
WHERE ssbe.name = N'MyServiceBrokerEndpoint';
```

When a route specifies **'LOCAL'** for the next_hop_address, the message is delivered to a service within the current instance of SQL Server.

When a route specifies **'TRANSPORT'** for the next_hop_address, the network address is determined based on the network address in the name of the service. A route that specifies **'TRANSPORT'** can specify a service name or broker instance.

When the next_hop_address is the principal server for a database mirror, you must also specify the MIRROR_ADDRESS for the mirror server. Otherwise, this route does not automatically failover to the mirror server.

Note

This option is not available in a contained database.

MIRROR_ADDRESS = 'next_hop_mirror_address'

Specifies the network address for the mirror server of a mirrored pair whose principal server is at the next_hop_address. The next_hop_mirror_address specifies a TCP/IP address in the following format:

TCP://{ *dns_name* | *netbios_name* | *ip_address* } **:** *port_number*

The specified *port_number* must match the port number for the Service Broker endpoint of an instance of SQL Server at the specified computer. This can be obtained by running the following query in the selected database:

```
SELECT tcpe.port
```

```
FROM sys.tcp_endpoints AS tcpe
INNER JOIN sys.service_broker_endpoints AS ssbe
   ON ssbe.endpoint_id = tcpe.endpoint_id
WHERE ssbe.name = N'MyServiceBrokerEndpoint';
```

When the MIRROR_ADDRESS is specified, the route must specify the SERVICE_NAME clause and the BROKER_INSTANCE clause. A route that specifies **'LOCAL'** or **'TRANSPORT'** for the next_hop_address might not specify a mirror address.

Note

This option is not available in a contained database.

Remarks

The routing table that stores the routes is a meta-data table that can be read through the **sys.routes** catalog view. The routing table can only be updated through the CREATE ROUTE, ALTER ROUTE, and DROP ROUTE statements.

Clauses that are not specified in the ALTER ROUTE command remain unchanged. Therefore, you cannot ALTER a route to specify that the route does not time out, that the route matches any service name, or that the route matches any broker instance. To change these characteristics of a route, you must drop the existing route and create a new route with the new information.

When a route specifies **'TRANSPORT'** for the next_hop_address, the network address is determined based on the name of the service. SQL Server can successfully process service names that begin with a network address in a format that is valid for a next_hop_address. Services with names that contain valid network addresses will route to the network address in the service name.

The routing table can contain any number of routes that specify the same service, network address, and/or broker instance identifier. In this case, Service Broker chooses a route using a procedure designed to find the most exact match between the information specified in the conversation and the information in the routing table.

To alter the AUTHORIZATION for a service, use the ALTER AUTHORIZATION statement.

Permissions

Permission for altering a route defaults to the owner of the route, members of the **db_ddladmin** or **db_owner** fixed database roles, and members of the **sysadmin** fixed server role.

Examples

A. Changing the service for a route

The following example modifies the `ExpenseRoute` route to point to the remote service `//Adventure-Works.com/Expenses`.

```
ALTER ROUTE ExpenseRoute
   WITH
```

```
    SERVICE_NAME = '//Adventure-Works.com/Expenses'
```

B. Changing the target database for a route

The following example changes the target database for the `ExpenseRoute` route to the database identified by the unique identifier `D8D4D268-00A3-4C62-8F91-634B89B1E317`.

```
ALTER ROUTE ExpenseRoute
   WITH
     BROKER_INSTANCE = 'D8D4D268-00A3-4C62-8F91-634B89B1E317'
```

C. Changing the address for a route

The following example changes the network address for the `ExpenseRoute` route to TCP port `1234` on the host with the IP address `10.2.19.72`.

```
ALTER ROUTE ExpenseRoute
   WITH
     ADDRESS = 'TCP://10.2.19.72:1234'
```

D. Changing the database and address for a route

The following example changes the network address for the `ExpenseRoute` route to TCP port `1234` on the host with the DNS name `www.Adventure-Works.com`. It also changes the target database to the database identified by the unique identifier `D8D4D268-00A3-4C62-8F91-634B89B1E317`.

```
ALTER ROUTE ExpenseRoute
   WITH
     BROKER_INSTANCE = 'D8D4D268-00A3-4C62-8F91-634B89B1E317',
     ADDRESS = 'TCP://www.Adventure-Works.com:1234'
```

See Also

CREATE ROUTE

DROP ROUTE

EVENTDATA

ALTER SCHEMA

Transfers a securable between schemas.

 Transact-SQL Syntax Conventions

Syntax

```
ALTER SCHEMA schema_name
  TRANSFER [ <entity_type> :: ] securable_name [;]
```

<entity_type> ::=

{

Object | Type | XML Schema Collection

}

Arguments

schema_name

Is the name of a schema in the current database, into which the securable will be moved. Cannot be SYS or INFORMATION_SCHEMA.

<entity_type>

Is the class of the entity for which the owner is being changed. Object is the default.

securable_name

Is the one-part or two-part name of a schema-contained securable to be moved into the schema.

Remarks

Users and schemas are completely separate.

ALTER SCHEMA can only be used to move securables between schemas in the same database. To change or drop a securable within a schema, use the ALTER or DROP statement specific to that securable.

If a one-part name is used for securable_name, the name-resolution rules currently in effect will be used to locate the securable.

All permissions associated with the securable will be dropped when the securable is moved to the new schema. If the owner of the securable has been explicitly set, the owner will remain unchanged. If the owner of the securable has been set to SCHEMA OWNER, the owner will remain SCHEMA OWNER; however, after the move SCHEMA OWNER will resolve to the owner of the new schema. The principal_id of the new owner will be NULL.

To change the schema of a table or view by using SQL Server Management Studio, in Object Explorer, right-click the table or view and then click **Design**. Press **F4** to open the Properties window. In the **Schema** box, select a new schema.

Caution

Beginning with SQL Server 2005, the behavior of schemas changed. As a result, code that assumes that schemas are equivalent to database users may no longer return correct results. Old catalog views, including , should not be used in a database in which any of the following DDL statements have ever been used: CREATE SCHEMA, ALTER SCHEMA, DROP SCHEMA, CREATE USER, ALTER USER, DROP USER, CREATE ROLE, ALTER ROLE, DROP ROLE, CREATE APPROLE, ALTER APPROLE, DROP APPROLE, ALTER AUTHORIZATION. In such databases you must instead use the new catalog views. The

new catalog views take into account the separation of principals and schemas that was introduced in SQL Server 2005. For more information about catalog views, see .

Permissions

To transfer a securable from another schema, the current user must have CONTROL permission on the securable (not schema) and ALTER permission on the target schema.

If the securable has an EXECUTE AS OWNER specification on it and the owner is set to SCHEMA OWNER, the user must also have IMPERSONATION permission on the owner of the target schema.

All permissions associated with the securable that is being transferred are dropped when it is moved.

Examples

A. Transferring ownership of a table

The following example modifies the schema `HumanResources` by transferring the table `Address` from schema `Person` into the schema.

```
USE AdventureWorks2012;
GO
ALTER SCHEMA HumanResources TRANSFER Person.Address;
GO
```

B. Transferring ownership of a type

The following example creates a type in the `Production` schema, and then transfers the type to the `Person` schema.

```
USE AdventureWorks2012;
GO

CREATE TYPE Production.TestType FROM [varchar](10) NOT NULL ;
GO

-- Check the type owner.
SELECT sys.types.name, sys.types.schema_id, sys.schemas.name
    FROM sys.types JOIN sys.schemas
        ON sys.types.schema_id = sys.schemas.schema_id
    WHERE sys.types.name = 'TestType' ;
GO
```

```
-- Change the type to the Person schema.
ALTER SCHEMA Person TRANSFER type::Production.TestType ;
GO

-- Check the type owner.
SELECT sys.types.name, sys.types.schema_id, sys.schemas.name
    FROM sys.types JOIN sys.schemas
        ON sys.types.schema_id = sys.schemas.schema_id
    WHERE sys.types.name = 'TestType' ;
GO
```

See Also

CREATE SCHEMA (Transact-SQL)

DROP SCHEMA (Transact-SQL)

eventdata (Transact-SQL)

ALTER SEARCH PROPERTY LIST

Adds a specified search property to, or drops it from the specified search property list.

Important
CREATE SEARCH PROPERTY LIST, ALTER SEARCH PROPERTY LIST, and DROP SEARCH PROPERTY LIST are supported only under compatibility level 110. Under lower compatibility levels, these statements are not supported.

Syntax

```
ALTER SEARCH PROPERTY LIST list_name
{
  ADD 'property_name'
    WITH
    (
       PROPERTY_SET_GUID = 'property_set_guid'
     , PROPERTY_INT_ID = property_int_id
    [ , PROPERTY_DESCRIPTION = 'property_description' ]
    )
 | DROP 'property_name'
}
```

;

Arguments

list_name

Is the name of the property list being altered. list_name is an identifier.

To view the names of the existing property lists, use the sys.registered_search_property_lists catalog view, as follows:

```
SELECT name FROM sys.registered_search_property_lists;
```

ADD

Adds a specified search property to the property list specified by list_name. The property is registered for the search property list . Before newly added properties can be used for property searching, the associated full-text index or indexes must be repopulated. For more information, see ALTER FULLTEXT INDEX (Transact-SQL).

Note

To add a given search property to a search property list, you must provide its property-set GUID (property_set_guid) and property int ID (property_int_id). For more information, see "Obtaining Property Set GUIDS and Identifiers," later in this topic.

property_name

Specifies the name to be used to identify the property in full-text queries. property_name must uniquely identify the property within the property set. A property name can contain internal spaces. The maximum length of property_name is 256 characters. This name can be a user-friendly name, such as Author or Home Address, or it can be the Windows canonical name of the property, such as **System.Author** or **System.Contact.HomeAddress**.

Developers will need to use the value you specify for property_name to identify the property in the CONTAINS predicate. Therefore, when adding a property it is important to specify a value that meaningfully represents the property defined by the specified property set GUID (property_set_guid) and property identifier (property_int_id). For more information about property names, see "Remarks," later in this topic.

To view the names of properties that currently exist in a search property list of the current database, use the sys.registered_search_properties catalog view, as follows:

```
SELECT property_name FROM sys.registered_search_properties;
```

PROPERTY_SET_GUID = 'property_set_guid'

Specifies the identifier of the property set to which the property belongs. This is a globally unique identifier (GUID). For information about obtaining this value, see "Remarks," later in this topic.

To view the property set GUID of any property that exists in a search property list of the current database, use the sys.registered_search_properties catalog view, as follows:

```
SELECT property_set_guid FROM
sys.registered_search_properties;
```

PROPERTY_INT_ID = property_int_id

Specifies the integer that identifies the property within its property set. For information about obtaining this value, see "Remarks."

To view the integer identifier of any property that exists in a search property list of the current database, use the sys.registered_search_properties catalog view, as follows:

```
SELECT property_int_id FROM sys.registered_search_properties;
```

Note

A given combination of property_set_guid and property_int_id must be unique in a search property list. If you try to add an existing combination, the ALTER SEARCH PROPERTY LIST operation fails and issues an error. This means that you can define only one name for a given property.

PROPERTY_DESCRIPTION = 'property_description'

Specifies a user-defined description of the property. property_description is a string of up to 512 characters. This option is optional.

DROP

Drops the specified property from the property list specified by list_name. Dropping a property unregisters it, so it is no longer searchable.

Remarks

Each full-text index can have only one search property list.

To enable querying on a given search property, you must add it to the search property list of the full-text index and then repopulate the index.

When specifying a property you can arrange the PROPERTY_SET_GUID, PROPERTY_INT_ID, and PROPERTY_DESCRIPTION clauses in any order, as a comma-separated list within parentheses, for example:

```
ALTER SEARCH PROPERTY LIST CVitaProperties
ADD 'System.Author'
WITH (
      PROPERTY_DESCRIPTION = 'Author or authors of a given document.',
      PROPERTY_SET_GUID   = 'F29F85E0-4FF9-1068-AB91-08002B27B3D9',
      PROPERTY_INT_ID = 4
      );
```

Note

This example uses the property name, `System.Author`, which is similar to the concept of canonical property names introduced in Windows Vista (Windows canonical name).

Obtaining Property Values

Full-text search maps a search property to a full-text index by using its property set GUID and property integer ID. For information about how to obtain these for properties that have been defined by Microsoft, see Find Property Set GUIDs and Property Integer IDs for Search Properties. For information about properties defined by an independent software vendor (ISV), see the documentation of that vendor.

Making Added Properties Searchable

Adding a search property to a search property list registers the property. A newly added property can be immediately specified in CONTAINS queries. However, property-scoped full-text queries on a newly added property will not return documents until the associated full-text index is repopulated. For example, the following property-scoped query on a newly added property, *new_search_property*, will not return any documents until the full-text index associated with the target table (*table_name*) is repopulated:

```
SELECT column_name FROM table_name WHERE CONTAINS( PROPERTY( column_name,
'new_search_property' ), 'contains_search_condition');
GO
```

To start a full population, use the following ALTER FULLTEXT INDEX (Transact-SQL) statement:

```
USE database_name;
GO
ALTER FULLTEXT INDEX ON table_name START FULL POPULATION;
GO
```

Note
Repopulation is not needed after a property is dropped from a property list, because only the properties that remain in the search property list are available for full-text querying.

Related References

To create a property list

- CREATE SEARCH PROPERTY LIST (Transact-SQL)

To drop a property list

- DROP SEARCH PROPERTY LIST (Transact-SQL)

To add or remove a property list on a full-text index

- ALTER FULLTEXT INDEX (Transact-SQL)

To run a population on a full-text index

- ALTER FULLTEXT INDEX (Transact-SQL)

Permissions

Requires CONTROL permission on the property list.

Examples

A. Adding a property

The following example adds several properties—`Title`, `Author`, and `Tags`—to a property list named `DocumentPropertyList`.

Note
For an example that creates `DocumentPropertyList` property list, see CREATE SEARCH PROPERTY LIST (Transact-SQL).

```
ALTER SEARCH PROPERTY LIST DocumentPropertyList
   ADD 'Title'
   WITH ( PROPERTY_SET_GUID = 'F29F85E0-4FF9-1068-AB91-08002B27B3D9',
PROPERTY_INT_ID = 2,
      PROPERTY_DESCRIPTION = 'System.Title - Title of the item.' );

ALTER SEARCH PROPERTY LIST DocumentPropertyList
    ADD 'Author'
   WITH ( PROPERTY_SET_GUID = 'F29F85E0-4FF9-1068-AB91-08002B27B3D9',
PROPERTY_INT_ID = 4,
      PROPERTY_DESCRIPTION = 'System.Author - Author or authors of the item.'
);

ALTER SEARCH PROPERTY LIST DocumentPropertyList
    ADD 'Tags'
   WITH ( PROPERTY_SET_GUID = 'F29F85E0-4FF9-1068-AB91-08002B27B3D9',
PROPERTY_INT_ID = 5,
      PROPERTY_DESCRIPTION = 'System.Keywords - Set of keywords (also known
as tags) assigned to the item.' );
```

Note
You must associate a given search property list with a full-text index before using it for property-scoped queries. To do so, use an ALTER FULLTEXT INDEX statement and specify the SET SEARCH PROPERTY LIST clause.

B. Dropping a property

The following example drops the `Comments` property from the `DocumentPropertyList` property list.

```
ALTER SEARCH PROPERTY LIST DocumentPropertyList
```

```
DROP 'Comments' ;
```

See Also

CREATE SEARCH PROPERTY LIST (Transact-SQL)

DROP SEARCH PROPERTY LIST (Transact-SQL)

sys.registered_search_properties (Transact-SQL)

sys.registered_search_property_lists (Transact-SQL)

sys.dm_fts_index_keywords_by_property (Transact-SQL)

Using Search Property Lists to Search for Properties (Full-Text Search)

Obtaining a Property Set GUID and Property Integer Identifier for a Search Property List (SQL Server)

ALTER SEQUENCE

Modifies the arguments of an existing sequence object. If the sequence was created with the **CACHE** option, altering the sequence will recreate the cache.

Sequences objects are created by using the CREATE SEQUENCE statement. Sequences are integer values and can be of any data type that returns an integer. The data type cannot be changed by using the ALTER SEQUENCE statement. To change the data type, drop and create the sequence object.

A sequence is a user-defined schema bound object that generates a sequence of numeric values according to a specification. New values are generated from a sequence by calling the NEXT VALUE FOR function. Use **sp_sequence_get_range** to get multiple sequence numbers at once. For information and scenarios that use both CREATE SEQUENCE, **sp_sequence_get_range**, and the NEXT VALUE FOR function, see Creating and Using Sequence Numbers.

 Transact-SQL Syntax Conventions

Syntax

```
ALTER SEQUENCE [schema_name. ] sequence_name
   [ RESTART [ WITH <constant> ] ]
   [ INCREMENT BY <constant> ]
   [ { MINVALUE <constant> } | { NO MINVALUE } ]
   [ { MAXVALUE <constant> } | { NO MAXVALUE } ]
   [ CYCLE | { NO CYCLE } ]
   [ { CACHE [ <constant> ] } | { NO CACHE } ]
   [ ; ]
```

Arguments

sequence_name

Specifies the unique name by which the sequence is known in the database. Type is **sysname**.

RESTART [WITH <constant>]

The next value that will be returned by the sequence object. If provided, the RESTART WITH value must be an integer that is less than or equal to the maximum and greater than or equal to the minimum value of the sequence object. If the WITH value is omitted, the sequence numbering restarts based on the original CREATE SEQUENCE options.

INCREMENT BY <constant>

The value that is used to increment (or decrement if negative) the sequence object's base value for each call to the NEXT VALUE FOR function. If the increment is a negative value the sequence object is descending, otherwise, it is ascending. The increment can not be 0.

[MINVALUE <constant> | NO MINVALUE]

Specifies the bounds for sequence object. If NO MINVALUE is specified, the minimum possible value of the sequence data type is used.

[MAXVALUE <constant> | NO MAXVALUE

Specifies the bounds for sequence object. If NO MAXVALUE is specified, the maximum possible value of the sequence data type is used.

[CYCLE | NO CYCLE]

This property specifies whether the sequence object should restart from the minimum value (or maximum for descending sequence objects) or throw an exception when its minimum or maximum value is exceeded.

Note

After cycling the next value is the minimum or maximum value, not the START VALUE of the sequence.

[CACHE [<constant>] | NO CACHE]

Increases performance for applications that use sequence objects by minimizing the number of IOs that are required to persist generated values to the system tables.

For more information about the behavior of the cache, see CREATE SEQUENCE (Transact-SQL).

Remarks

For information about how sequences are created and how the sequence cache is managed, see CREATE SEQUENCE (Transact-SQL).

The MINVALUE for ascending sequences and the MAXVALUE for descending sequences cannot be altered to a value that does not permit the START WITH value of the sequence. To change the MINVALUE of an ascending sequence to a number larger than the START WITH value or to change the MAXVALUE of a descending sequence to a number smaller than the START WITH value, include the RESTART WITH argument to restart the sequence at a desired point that falls within the minimum and maximum range.

Metadata

For information about sequences, query sys.sequences.

Security

Permissions

Requires **ALTER** permission on the sequence or **ALTER** permission on the schema. To grant **ALTER** permission on the sequence, use **ALTER ON OBJECT** in the following format:

```
GRANT ALTER ON OBJECT::Test.TinySeq TO [AdventureWorks\Larry]
```

The ownership of a sequence object can be transferred by using the **ALTER AUTHORIZATION** statement.

Audit

To audit **ALTER SEQUENCE**, monitor the **SCHEMA_OBJECT_CHANGE_GROUP**.

Examples

For examples of both creating sequences and using the **NEXT VALUE FOR** function to generate sequence numbers, see Creating and Using Sequence Numbers.

A. Altering a sequence

The following example creates a schema named Test and a sequence named TestSeq using the **int** data type, having a range from 0 to 255. The sequence starts with 125 and increments by 25 every time that a number is generated. Because the sequence is configure to cycle, when the value exceeds the maximum value of 200, the sequence restarts at the minimum value of 100.

```
CREATE SCHEMA Test ;
GO

CREATE SEQUENCE Test.TestSeq
    AS int
    START WITH 125
    INCREMENT BY 25
    MINVALUE 100
    MAXVALUE 200
    CYCLE
    CACHE 3
;
GO
```

The following example alters the TestSeq sequence to have a range from 0 to 255. The sequence restarts the numbering series with 100 and increments by 50 every time that a number is generated.

```
ALTER SEQUENCE Test. TestSeq
    RESTART WITH 100
    INCREMENT BY 50
    MINVALUE 50
    MAXVALUE 200
    NO CYCLE
    NO CACHE
;
GO
```

Because the sequence will not cycle, the **NEXT VALUE FOR** function will result in an error when the sequence exceeds 200.

B. Restarting a sequence

The following example creates a sequence named CountBy1. The sequence uses the default values.

```
CREATE SEQUENCE Test.CountBy1 ;
```

To generate a sequence value, the owner then executes the following statement:

```
SELECT NEXT VALUE FOR Test.CountBy1
```

The value returned of -9,223,372,036,854,775,808 is the lowest possible value for the **bigint** data type. The owner realizes he wanted the sequence to start with 1, but did not indicate the **START WITH** clause when he created the sequence. To correct this error, the owner executes the following statement.

```
ALTER SEQUENCE Test.CountBy1 RESTART WITH 1 ;
```

Then the owner executes the following statement again to generate a sequence number.

```
SELECT NEXT VALUE FOR Test.CountBy1;
```

The number is now 1, as expected.

The CountBy1 sequence was created using the default value of NO CYCLE so it will stop operating after generating number 9,223,372,036,854,775,807. Subsequent calls to the sequence object will return error 11728. The following statement changes the sequence object to cycle and sets a cache of 20.

```
ALTER SEQUENCE Test.CountBy1
    CYCLE
    CACHE 20 ;
```

Now when the sequence object reaches 9,223,372,036,854,775,807 it will cycle, and the next number after cycling will be the minimum of the data type, -9,223,372,036,854,775,808.

The owner realized that the **bigint** data type uses 8 bytes each time it is used. The **int** data type that uses 4 bytes is sufficient. However the data type of a sequence object cannot be altered. To change to an **int** data type, the owner must drop the sequence object and recreate the object with the correct data type.

See Also

CREATE SEQUENCE (Transact-SQL)

DROP SEQUENCE (Transact-SQL)

NEXT VALUE FOR function (Transact-SQL)

Creating and Using Sequence Numbers

sp_sequence_get_range (Transact-SQL)

ALTER SERVER AUDIT

Alters a server audit object using the SQL Server Audit feature. For more information, see Understanding SQL Server Audit.

Transact-SQL Syntax Conventions

Syntax

```
ALTER SERVER AUDIT audit_name
{
   [ TO { { FILE ( <file_options> [, ...n] ) } | APPLICATION_LOG | SECURITY_LOG } ]
   [ WITH ( <audit_options> [ , ...n] ) ]
   [ WHERE <predicate_expression> ]
}
| REMOVE WHERE
| MODIFY NAME = new_audit_name
[ ; ]

<file_options>::=
{
    FILEPATH = 'os_file_path'
   | MAXSIZE = { max_size { MB | GB | TB } | UNLIMITED }
   | MAX_ROLLOVER_FILES = { integer | UNLIMITED }
   | MAX_FILES = integer
```

| RESERVE_DISK_SPACE = { ON | OFF }
}

<audit_options>::=

{
QUEUE_DELAY = **integer**
| ON_FAILURE = { CONTINUE | SHUTDOWN | FAIL_OPERATION }
| STATE = = { ON | OFF }
}

<predicate_expression>::=

{
[**NOT**] <predicate_factor>
[{ **AND** | **OR** } [**NOT**] { <predicate_factor> }]
[,...n]
}

<predicate_factor>::=

event_field_name { = | **<** **>** | **!** = | **>** | **>** = | **<** | **<** = } { **number** | **'** **string** **'** }

Arguments

TO { FILE | APPLICATION_LOG | SECURITY }

Determines the location of the audit target. The options are a binary file, the Windows application log, or the Windows security log.

FILEPATH = 'os_file_path'

The path of the audit trail. The file name is generated based on the audit name and audit GUID.

MAXSIZE = max_size

Specifies the maximum size to which the audit file can grow. The max_size value must be an integer followed by **MB**, **GB**, **TB**, or **UNLIMITED**. The minimum size that you can specify for max_size is 2 **MB** and the maximum is 2,147,483,647 **TB**. When **UNLIMITED** is specified the file grows until the disk is full. Specifying a value lower than 2 MB will raise the error MSG_MAXSIZE_TOO_SMALL. The default value is **UNLIMITED**.

MAX_ROLLOVER_FILES = integer | UNLIMITED

Specifies the maximum number of files to retain in the file system. When the setting of MAX_ROLLOVER_FILES=0 there is no limit imposed on the number of rollover files that will be created. The default value is 0. The maximum number of files that can be specified is

2,147,483,647.

MAX_FILES = integer

Specifies the maximum number of audit files that can be created. Does not rollover to the first file when the limit is reached. When the MAX_FILES limit is reached, any action that causes additional audit events to be generated will fail with an error.

RESERVE_DISK_SPACE = { ON | OFF }

This option pre-allocates the file on the disk to the MAXSIZE value. Only applies if MAXSIZE is not equal to UNLIMITED. The default value is OFF.

QUEUE_DELAY = integer

Determines the time in milliseconds that can elapse before audit actions are forced to be processed. A value of 0 indicates synchronous delivery. The minimum settable query delay value is 1000 (1 second), which is the default. The maximum is 2,147,483,647 (2,147,483.647 seconds or 24 days, 20 hours, 31 minutes, 23.647 seconds). Specifying an invalid number will raise the error MSG_INVALID_QUEUE_DELAY.

ON_FAILURE = { CONTINUE | SHUTDOWN | FAIL_OPERATION}

Indicates whether the instance writing to the target should fail, continue, or stop if SQL Server cannot write to the audit log.

CONTINUE

SQL Server operations continue. Audit records are not retained. The audit continues to attempt to log events and will resume if the failure condition is resolved. Selecting the continue option can allow unaudited activity which could violate your security policies. Use this option, when continuing operation of the Database Engine is more important than maintaining a complete audit.

SHUTDOWN

Forces a server shut down when the server instance writing to the target cannot write data to the audit target. The login issuing this must have the **SHUTDOWN** permission. If the logon does not have this permission, this function will fail and an error message will be raised. No audited events occur. Use the option when an audit failure could compromise the security or integrity of the system.

FAIL_OPERATION

Database actions fail if they cause audited events. Actions which do not cause audited events can continue, but no audited events can occur. The audit continues to attempt to log events and will resume if the failure condition is resolved. Use this option when maintaining a complete audit is more important than full access to the Database Engine.

STATE = { ON | OFF }

Enables or disables the audit from collecting records. Changing the state of a running audit (from ON to OFF) creates an audit entry that the audit was stopped, the principal that stopped the audit, and the time the audit was stopped.

MODIFY NAME = new_audit_name

Changes the name of the audit. Cannot be used with any other option.

predicate_expression

Specifies the predicate expression used to determine if an event should be processed or not. Predicate expressions are limited to 3000 characters, which limits string arguments.

event_field_name

Is the name of the event field that identifies the predicate source. Audit fields are described in fn_get_audit_file (Transact-SQL). All fields can be audited except `file_name` and `audit_file_offset`.

number

Is any numeric type including **decimal**. Limitations are the lack of available physical memory or a number that is too large to be represented as a 64-bit integer.

' string '

Either an ANSI or Unicode string as required by the predicate compare. No implicit string type conversion is performed for the predicate compare functions. Passing the wrong type results in an error.

Remarks

You must specify at least one of the TO, WITH, or MODIFY NAME clauses when you call ALTER AUDIT.

You must set the state of an audit to the OFF option in order to make changes to an audit. If ALTER AUDIT is run when an audit is enabled with any options other than STATE=OFF, you will receive a MSG_NEED_AUDIT_DISABLED error message.

You can add, alter, and remove audit specifications without stopping an audit.

You cannot change an audit's GUID after the audit has been created.

Permissions

To create, alter, or drop a server audit principal, you must have ALTER ANY SERVER AUDIT or the CONTROL SERVER permission.

Examples

A. Changing a server audit name

The following example changes the name of the server audit `HIPPA_Audit` to `HIPAA_Audit_Old`.

```
USE master
GO
ALTER SERVER AUDIT HIPAA_Audit
WITH (STATE = OFF);
```

```
GO
ALTER SERVER AUDIT HIPAA_Audit
MODIFY NAME = HIPAA_Audit_Old;
GO
ALTER SERVER AUDIT HIPAA_Audit_Old
WITH (STATE = ON);
GO
```

B. Changing a server audit target

The following example changes the server audit called `HIPPA_Audit` to a file target.

```
USE master
GO
ALTER SERVER AUDIT HIPAA_Audit
WITH (STATE = OFF);
GO
ALTER SERVER AUDIT HIPAA_Audit
TO FILE (FILEPATH ='\\SQLPROD_1\Audit\',
          MAXSIZE = 1000 MB,
          RESERVE_DISK_SPACE=OFF)
WITH (QUEUE_DELAY = 1000,
       ON_FAILURE = CONTINUE);
GO
ALTER SERVER AUDIT HIPAA_Audit
WITH (STATE = ON);
GO
```

C. Changing a server audit WHERE clause

The following example modifies the where clause created in example C of CREATE SERVER AUDIT (Transact-SQL). The new WHERE clause filters for the user defined event if of 27.

```
ALTER SERVER AUDIT [FilterForSensitiveData] WITH (STATE = OFF)
GO
ALTER SERVER AUDIT [FilterForSensitiveData]
WHERE user_defined_event_id = 27;
GO
ALTER SERVER AUDIT [FilterForSensitiveData] WITH (STATE = ON);
```

```
GO
```

D. Removing a WHERE clause

The following example removes a WHERE clause predicate expression.

```
ALTER SERVER AUDIT [FilterForSensitiveData] WITH (STATE = OFF)
GO
ALTER SERVER AUDIT [FilterForSensitiveData]
REMOVE WHERE;
GO
ALTER SERVER AUDIT [FilterForSensitiveData] WITH (STATE = ON);
GO
```

E. Renaming a server audit

The following example changes the server audit name from `FilterForSensitiveData` to `AuditDataAccess`.

```
ALTER SERVER AUDIT [FilterForSensitiveData] WITH (STATE = OFF)
GO
ALTER SERVER AUDIT [FilterForSensitiveData]
MODIFY NAME = AuditDataAccess;
GO
ALTER SERVER AUDIT [AuditDataAccess] WITH (STATE = ON);
GO
```

See Also

DROP SERVER AUDIT (Transact-SQL)
CREATE SERVER AUDIT SPECIFICATION (Transact-SQL)
ALTER SERVER AUDIT SPECIFICATION (Transact-SQL)
DROP SERVER AUDIT SPECIFICATION (Transact-SQL)
CREATE DATABASE AUDIT SPECIFICATION (Transact-SQL)
ALTER DATABASE AUDIT SPECIFICATION (Transact-SQL)
DROP DATABASE AUDIT SPECIFICATION (Transact-SQL)
ALTER AUTHORIZATION (Transact-SQL)
fn_get_audit_file (Transact-SQL)
sys.server_audits (Transact-SQL)
sys.server_file_audits (Transact-SQL)
sys.server_audit_specifications (Transact-SQL)
sys.server_audit_specifications_details (Transact-SQL)

sys.database_audit_specifications (Transact-SQL)

sys.audit_database_specification_details (Transact-SQL)

sys.dm_server_audit_status

sys.dm_audit_actions

Create a Server Audit and Server Audit Specification

ALTER SERVER AUDIT SPECIFICATION

Alters a server audit specification object using the SQL Server Audit feature. For more information, see Understanding SQL Server Audit.

 Transact-SQL Syntax Conventions

Syntax

```
ALTER SERVER AUDIT SPECIFICATION audit_specification_name
{
   [ FOR SERVER AUDIT audit_name ]
   [ { { ADD | DROP } ( audit_action_group_name )
     } [, ...n] ]
   [ WITH ( STATE = { ON | OFF } ) ]
}
[ ; ]
```

Arguments

audit_specification_name

The name of the audit specification.

audit_name

The name of the audit to which this specification is applied.

audit_action_group_name

Name of a group of server-level auditable actions. For a list of Audit Action Groups, see SQL Server Audit Action Groups and Actions.

WITH (STATE = { ON | OFF })

Enables or disables the audit from collecting records for this audit specification.

Remarks

You must set the state of an audit specification to the OFF option to make changes to an audit specification. If ALTER SERVER AUDIT SPECIFICATION is executed when an audit specification is enabled with any options other than STATE=OFF, you will receive an error message.

Permissions

Users with the ALTER ANY SERVER AUDIT permission can alter server audit specifications and bind them to any audit.

After a server audit specification is created, it can be viewed by principals with the CONTROL SERVER, or ALTER ANY SERVER AUDIT permissions, the sysadmin account, or principals having explicit access to the audit.

Examples

The following example creates a server audit specification called `HIPPA_Audit_Specification`. It drops the audit action group for failed logins, and adds an audit action group for Database Object Access for a SQL Server audit called `HIPPA_Audit`.

```
ALTER SERVER AUDIT SPECIFICATION HIPPA_Audit_Specification
FOR SERVER AUDIT HIPPA_Audit
    DROP (FAILED_LOGIN_GROUP)
    ADD (DATABASE_OBJECT_ACCESS_GROUP);
GO
```

For a full example about how to create an audit, see Understanding SQL Server Audit.

Updated content
Corrected the Permissions section.

See Also

CREATE SERVER AUDIT (Transact-SQL)
ALTER SERVER AUDIT (Transact-SQL)
DROP SERVER AUDIT (Transact-SQL)
CREATE SERVER AUDIT SPECIFICATION (Transact-SQL)
DROP SERVER AUDIT SPECIFICATION (Transact-SQL)
CREATE DATABASE AUDIT SPECIFICATION (Transact-SQL)
ALTER DATABASE AUDIT SPECIFICATION (Transact-SQL)
DROP DATABASE AUDIT SPECIFICATION (Transact-SQL)
ALTER AUTHORIZATION (Transact-SQL)
fn_get_audit_file (Transact-SQL)
sys.server_audits (Transact-SQL)
sys.server_file_audits (Transact-SQL)
sys.server_audit_specifications (Transact-SQL)
sys.server_audit_specifications_details (Transact-SQL)

sys.database_audit_specifications (Transact-SQL)
sys.audit_database_specification_details (Transact-SQL)
sys.dm_server_audit_status
sys.dm_audit_actions
Create a Server Audit and Server Audit Specification

ALTER SERVER CONFIGURATION

Modifies global configuration settings for the current server in SQL Server 2012.

 Transact-SQL Syntax Conventions

Syntax

```
ALTER SERVER CONFIGURATION
SET <optionspec>

<optionspec> ::=
{
    <process_affinity>
  | <diagnostic_log>
  | <failover_cluster_property>
}

<process_affinity> ::=
  PROCESS AFFINITY
{
   CPU = { AUTO | <CPU_range_spec> }
 | NUMANODE = <NUMA_node_range_spec>
}
  <CPU_range_spec> ::=
  { CPU_ID | CPU_ID  TO CPU_ID } [ ,...n ]

  <NUMA_node_range_spec> ::=
  { NUMA_node_ID | NUMA_node_ID TO NUMA_node_ID } [ ,...n ]

<diagnostic_log> ::=
  DIAGNOSTICS LOG
```

```
  {
    ON
  | OFF
  | PATH = { 'os_file_path' | DEFAULT }
  | MAX_SIZE = { 'log_max_size' MB | DEFAULT }
  | MAX_FILES = { 'max_file_count' | DEFAULT }
  }

<failover_cluster_property> ::=
  FAILOVER CLUSTER PROPERTY <resource_property>

  <resource_property>::=
{
    VerboseLogging = { 'logging_detail' | DEFAULT }
  | SqlDumperDumpFlags = { 'dump_file_type' | DEFAULT }
  | SqlDumperDumpPath = { 'os_file_path'| DEFAULT }
  | SqlDumperDumpTimeOut = { 'dump_time-out' | DEFAULT }
  | FailureConditionLevel = { 'failure_condition_level' | DEFAULT }
  | HealthCheckTimeout = { 'health_check_time-out' | DEFAULT }
}
```

Arguments

<process_affinity> ::=

PROCESS AFFINITY

Enables hardware threads to be associated with CPUs.

CPU = { AUTO | <CPU_range_spec> }

Distributes SQL Server worker threads to each CPU within the specified range. CPUs outside the specified range will not have assigned threads.

AUTO

Specifies that no thread is assigned a CPU. The operating system can freely move threads among CPUs based on the server workload. This is the default and recommended setting.

<CPU_range_spec> ::=

Specifies the CPU or range of CPUs to assign threads to.

{ CPU_ID | CPU_ID TO CPU_ID } [,...n]

Is the list of one or more CPUs. CPU IDs begin at 0 and are **integer** values.

NUMANODE = <NUMA_node_range_spec>

Assigns threads to all CPUs that belong to the specified NUMA node or range of nodes.

<NUMA_node_range_spec> ::=

Specifies the NUMA node or range of NUMA nodes.

{ NUMA_node_ID | NUMA_node_ID TO NUMA_node_ID } [,...n]

Is the list of one or more NUMA nodes. NUMA node IDs begin at 0 and are **integer** values.

<diagnostic_log> ::=

DIAGNOSTICS LOG

Starts or stops logging diagnostic data captured by the sp_server_diagnostics procedure, and sets SQLDIAG log configuration parameters such as the log file rollover count, log file size, and file location. For more information, see How to: View and Read SQL Server Failover Cluster Diagnostics Log.

ON

Starts SQL Server logging diagnostic data in the location specified in the PATH file option. This is the default.

OFF

Stops logging diagnostic data.

PATH = { 'os_file_path' | DEFAULT }

Path indicating the location of the diagnostic logs. The default location is <\MSSQL\Log> within the installation folder of the SQL Server failover cluster instance.

MAX_SIZE = { 'log_max_size' MB | DEFAULT }

Maximum size in megabytes to which each diagnostic log can grow. The default is 100 MB.

MAX_FILES = { 'max_file_count' | DEFAULT }

Maximum number of diagnostic log files that can be stored on the computer before they are recycled for new diagnostic logs.

<failover_cluster_property> ::=

FAILOVER CLUSTER PROPERTY

Modifies the SQL Server resource private failover cluster properties.

VERBOSE LOGGING = { 'logging_detail' | DEFAULT }

Sets the logging level for SQL Server Failover Clustering. It can be turned on to provide additional details in the error logs for troubleshooting.

- 0 – Logging is turned off (default)
- 1 - Errors only
- 2 – Errors and warnings

SQLDUMPEREDUMPFLAGS

Determines the type of dump files generated by SQL Server SQLDumper utility. The default setting is 0. For more information, see SQL Server Dumper Utility Knowledgebase article.

SQLDUMPERDUMPPATH = { 'os_file_path'| DEFAULT }

The location where the SQLDumper utility stores the dump files. For more information, see SQL Server Dumper Utility Knowledgebase article.

SQLDUMPERDUMPTIMEOUT = { 'dump_time-out' | DEFAULT }

The time-out value in milliseconds for the SQLDumper utility to generate a dump in case of a SQL Server failure. The default value is 0, which means there is no time limit to complete the dump. For more information, see SQL Server Dumper Utility Knowledgebase article.

FAILURECONDITIONLEVEL = { 'failure_condition_level' | DEFAULT }

Tthe conditions under which the SQL Server failover cluster instance should failover or restart. The default value is 3, which means that the SQL Server resource will failover or restart on critical server errors. For more information about this and other failure condition levels, see How to: Configure FailureConditionLevel Property Settings.

HEALTHCHECKTIMEOUT = { 'health_check_time-out' | DEFAULT }

The time-out value for how long the SQL Server Database Engine resource DLL should wait for the server health information before it considers the instance of SQL Server as unresponsive. The time-out value is expressed in milliseconds. The default is 60000 milliseconds (60 seconds).

General Remarks

This statement does not require a restart of SQL Server. In the case of a SQL Server failover cluster instance, it does not require a restart of the SQL Server cluster resource.

Limitations and Restrictions

This statement does not support DDL triggers.

Permissions

Requires ALTER SETTINGS permissions for the process affinity option; and ALTER SETTINGS and VIEW SERVER STATE permissions for the diagnostic log and failover cluster property options.

The SQL Server Database Engine resource DLL runs under the Local System account. Therefore, the Local System account must have read and write access to the specified path in the Diagnostic Log option.

Examples

Category	Featured syntax elements
Setting process affinity	CPU • NUMANODE • AUTO
Setting diagnostic log options	ON • OFF • PATH • MAX_SIZE
Setting failover cluster properties	HealthCheckTimeout

Setting process affinity

The examples in this section show how to set process affinity to CPUs and NUMA nodes. The examples assume that the server contains 256 CPUs that are arranged into four groups of 16 NUMA nodes each. Threads are not assigned to any NUMA node or CPU.

- Group 0: NUMA nodes 0 through 3, CPUs 0 to 63
- Group 1: NUMA nodes 4 through 7, CPUs 64 to 127
- Group 2: NUMA nodes 8 through 12, CPUs 128 to 191
- Group 3: NUMA nodes 13 through 16, CPUs 192 to 255

A. Setting affinity to all CPUs in groups 0 and 2

The following example sets affinity to all the CPUs in groups 0 and 2.

```
ALTER SERVER CONFIGURATION
SET PROCESS AFFINITY CPU=0 TO 63, 128 TO 191;
```

B. Setting affinity to all CPUs in NUMA nodes 0 and 7

The following example sets the CPU affinity to nodes `0` and `7` only.

```
ALTER SERVER CONFIGURATION
SET PROCESS AFFINITY NUMANODE=0, 7;
```

C. Setting affinity to CPUs 60 through 200

The following example sets affinity to CPUs 60 through 200.

```
ALTER SERVER CONFIGURATION
SET PROCESS AFFINITY CPU=60 TO 200;
```

D. Setting affinity to CPU 0 on a system that has two CPUs

The following example sets the affinity to `CPU=0` on a computer that has two CPUs. Before the following statement is executed the internal affinity bitmask is 00.

```
ALTER SERVER CONFIGURATION SET PROCESS AFFINITY CPU=0;
```

E. Setting affinity to AUTO

The following example sets affinity to `AUTO`.

```
ALTER SERVER CONFIGURATION
```

```
SET PROCESS AFFINITY CPU=AUTO;
```

Setting diagnostic log options

The examples in this section show how to set the values for the diagnostic log option.

A. Starting diagnostic logging

The following example starts the logging of diagnostic data.

```
ALTER SERVER CONFIGURATION SET DIAGNOSTICS LOG ON;
```

B. Stopping diagnostic logging

The following example stops the logging of diagnostic data.

```
ALTER SERVER CONFIGURATION SET DIAGNOSTICS LOG OFF;
```

C. Specifying the location of the diagnostic logs

The following example sets the location of the diagnostic logs to the specified file path.

```
ALTER SERVER CONFIGURATION
SET DIAGNOSTICS LOG PATH = 'C:\logs';
```

D. Specifying the maximum size of each diagnostic log

The following example set the maximum size of each diagnostic log to 10 megabytes.

```
ALTER SERVER CONFIGURATION
SET DIAGNOSTICS LOG MAX_SIZE = 10 MB;
```

Setting failover cluster properties

The following example illustrates setting the values of the SQL Server failover cluster resource properties.

A. Specifying the value for the HealthCheckTimeout property

The following example sets the `HealthCheckTimeout` option to 15,000 milliseconds (15 seconds).

```
ALTER SERVER CONFIGURATION
SET FAILOVER CLUSTER PROPERTY HealthCheckTimeout = 15000;
```

See Also

How to: Configure SQL Server to Use Soft-NUMA

sys.dm_os_schedulers (Transact-SQL)

sys.dm_os_memory_nodes (Transact-SQL)

ALTER SERVER ROLE

Changes the membership of a server role or changes name of a user-defined server role. Fixed server roles cannot be renamed.

Transact-SQL Syntax Conventions

Syntax

```
ALTER SERVER ROLE server_role_name
{
    [ ADD MEMBER server_principal ]
  | [ DROP MEMBER server_principal ]
  | [ WITH NAME = new_server_role_name ]
} [ ; ]
```

Arguments

server_role_name

Is the name of the server role to be changed.

ADD MEMBER server_principal

Adds the specified server principal to the server role. *server_principal* can be a login or a user-defined server role. *server_principal* cannot be a fixed server role, a database role, or sa.

DROP MEMBER server_principal

Removes the specified server principal from the server role. *server_principal* can be a login or a user-defined server role. *server_principal* cannot be a fixed server role, a database role, or sa.

WITH NAME = new_server_role_name

Specifies the new name of the user-defined server role. This name cannot already exist in the server.

Remarks

Changing the name of a user-defined server role does not change ID number, owner, or permissions of the role.

For changing role membership, ALTER SERVER ROLE replaces sp_addsrvrolemember and sp_dropsrvrolemember. These stored procedures are deprecated.

You can view server roles by querying the sys.server_role_members and sys.server_principals catalog views.

To change the owner of a user-defined server role, use ALTER AUTHORIZATION (Transact-SQL).

Permissions

Requires ALTER ANY SERVER ROLE permission on the server to change the name of a user-defined server role.

Fixed server roles

To add a member to a fixed server role, you must be a member of that fixed server role, or be a member of the sysadmin fixed server role.

Note
The CONTROL SERVER and ALTER ANY SERVER ROLE permissions are not sufficient to execute ALTER SERVER ROLE for a fixed server role, and ALTER permission cannot be granted on a fixed server role.

User-defined server roles

To add a member to a user-defined server role, you must be a member of the sysadmin fixed server role or have CONTROL SERVER or ALTER ANY SERVER ROLE permission. Or you must have ALTER permission on that role.

Note
Unlike fixed server roles, members of a user-defined server role do not inherently have permission to add members to that same role.

Examples

A. Changing the name of a server role

The following example creates a server role named `Product`, and then changes the name of server role to `Production`.

```
CREATE SERVER ROLE Product ;

ALTER SERVER ROLE Product WITH NAME = Production ;

GO
```

B. Adding a domain account to a server role

The following example adds a domain account named `adventure-works\roberto0` to the user-defined server role named `Production`.

```
ALTER SERVER ROLE Production ADD MEMBER [adventure-works\roberto0] ;
```

C. Adding a SQL Server login to a server role

The following example adds a SQL Server login named `Ted` to the `diskadmin` fixed server role.

```
ALTER SERVER ROLE diskadmin ADD MEMBER Ted ;

GO
```

D. Removing a domain account from a server role

The following example removes a domain account named `adventure-works\roberto0` from the user-defined server role named `Production`.

```
ALTER SERVER ROLE Production DROP MEMBER [adventure-works\roberto0] ;
```

E. Removing a SQL Server login from a server role

The following example removes the SQL Server login `Ted` from the `diskadmin` fixed server role.

```
ALTER SERVER ROLE Production DROP MEMBER Ted ;
GO
```

F. Granting a login the permission to add logins to a user-defined server role

The following example allows `Ted` to add other logins to the user-defined server role named `Production`.

```
GRANT ALTER ON SERVER ROLE::Production TO Ted ;
GO
```

G. To view role membership

To view role membership, use the **Server Role (Members)** page in SQL Server Management Studio or execute the following query:

```
SELECT SRM.role_principal_id, SP.name AS Role_Name,
SRM.member_principal_id, SP2.name  AS Member_Name
FROM sys.server_role_members AS SRM
JOIN sys.server_principals AS SP
    ON SRM.Role_principal_id = SP.principal_id
JOIN sys.server_principals AS SP2
    ON SRM.member_principal_id = SP2.principal_id
ORDER BY  SP.name,  SP2.name
```

See Also

CREATE SERVER ROLE (Transact-SQL)
DROP SERVER ROLE (Transact-SQL)
CREATE ROLE (Transact-SQL)
ALTER ROLE (Transact-SQL)
DROP ROLE (Transact-SQL)
Security Stored Procedures (Transact-SQL)
Security Functions (Transact-SQL)
Principals
sys.server_role_members (Transact-SQL)
sys.server_principals (Transact-SQL)

ALTER SERVICE

Changes an existing service.

 Transact-SQL Syntax Conventions

Syntax

```
ALTER SERVICE service_name
  [ ON QUEUE [ schema_name . ]queue_name ]
  [ ( < opt_arg > [ , ...n ] ) ]
[ ; ]

<opt_arg> ::=
  ADD CONTRACT contract_name | DROP CONTRACT contract_name
```

Arguments

service_name

Is the name of the service to change. Server, database, and schema names cannot be specified.

ON QUEUE [schema_name.] queue_name

Specifies the new queue for this service. Service Broker moves all messages for this service from the current queue to the new queue.

ADD CONTRACT contract_name

Specifies a contract to add to the contract set exposed by this service.

DROP CONTRACT contract_name

Specifies a contract to delete from the contract set exposed by this service. Service Broker sends an error message on any existing conversations with this service that use this contract.

Remarks

When the ALTER SERVICE statement deletes a contract from a service, the service can no longer be a target for conversations that use that contract. Therefore, Service Broker does not allow new conversations to the service on that contract. Existing conversations that use the contract are unaffected.

To alter the AUTHORIZATION for a service, use the ALTER AUTHORIZATION statement.

Permissions

Permission for altering a service defaults to the owner of the service, members of the **db_ddladmin** or **db_owner** fixed database roles, and members of the **sysadmin** fixed server role.

Examples

A. Changing the queue for a service

The following example changes the `//Adventure-Works.com/Expenses` service to use the queue `NewQueue`.

```
ALTER SERVICE [//Adventure-Works.com/Expenses]
    ON QUEUE NewQueue ;
```

B. Adding a new contract to the service

The following example changes the `//Adventure-Works.com/Expenses` service to allow dialogs on the contract `//Adventure-Works.com/Expenses`.

```
ALTER SERVICE [//Adventure-Works.com/Expenses]
    (ADD CONTRACT [//Adventure-Works.com/Expenses/ExpenseSubmission]) ;
```

C. Adding a new contract to the service, dropping existing contract

The following example changes the `//Adventure-Works.com/Expenses` service to allow dialogs on the contract `//Adventure-Works.com/Expenses/ExpenseProcessing` and to disallow dialogs on the contract `//Adventure-Works.com/Expenses/ExpenseSubmission`.

```
ALTER SERVICE [//Adventure-Works.com/Expenses]
    (ADD CONTRACT [//Adventure-Works.com/Expenses/ExpenseProcessing],
     DROP CONTRACT [//Adventure-Works.com/Expenses/ExpenseSubmission]) ;
```

See Also

DROP SERVICE (Transact-SQL)
DROP SERVICE
EVENTDATA

ALTER SERVICE MASTER KEY

Changes the service master key of an instance of SQL Server.

 Transact-SQL Syntax Conventions

Syntax

```
ALTER SERVICE MASTER KEY
   [ { <regenerate_option> | <recover_option> } ] [;]

<regenerate_option> ::=
   [ FORCE ] REGENERATE

<recover_option> ::=
   { WITH OLD_ACCOUNT = 'account_name' , OLD_PASSWORD = 'password' }
   |
   { WITH NEW_ACCOUNT = 'account_name' , NEW_PASSWORD = 'password' }
```

Arguments

FORCE

Indicates that the service master key should be regenerated, even at the risk of data loss. For more information, see Changing the SQL Server Service Account later in this topic.

REGENERATE

Indicates that the service master key should be regenerated.

OLD_ACCOUNT = 'account_name'

Specifies the name of the old Windows service account.

Warning

This option is obsolete. Do not use. Use SQL Server Configuration Manager instead.

OLD_PASSWORD = 'password'

Specifies the password of the old Windows service account.

Warning

This option is obsolete. Do not use. Use SQL Server Configuration Manager instead.

NEW_ACCOUNT = 'account_name'

Specifies the name of the new Windows service account.

Warning

This option is obsolete. Do not use. Use SQL Server Configuration Manager instead.

NEW_PASSWORD = 'password'

Specifies the password of the new Windows service account.

Warning

This option is obsolete. Do not use. Use SQL Server Configuration Manager instead.

Remarks

The service master key is automatically generated the first time it is needed to encrypt a linked server password, credential, or database master key. The service master key is encrypted using the local machine key or the Windows Data Protection API. This API uses a key that is derived from the Windows credentials of the SQL Server service account.

The service master key can only be decrypted by the service account under which it was created or by a principal that has access to the Windows credentials of that service account. Therefore, if you change the Windows account under which the SQL Server service runs, you must also enable decryption of the service master key by the new account.

SQL Server 2012 uses the AES encryption algorithm to protect the service master key (SMK) and the database master key (DMK). AES is a newer encryption algorithm than 3DES used in earlier versions. After upgrading an instance of the Database Engine to SQL Server 2012 the SMK and

DMK should be regenerated in order to upgrade the master keys to AES. For more information about regenerating the DMK, see ALTER MASTER KEY (Transact-SQL).

Changing the SQL Server Service Account

To change the SQL Server service account, use SQL Server Configuration Manager. To manage a change of the service account, SQL Server stores a redundant copy of the service master key protected by the machine account that has the necessary permissions granted to the SQL Server service group. If the computer is rebuilt, the same domain user that was previously used by the service account can recover the service master key. This does not work with local accounts or the Local System, Local Service, or Network Service accounts. When you are moving SQL Server to another computer, migrate the service master key by using backup and restore.

The REGENERATE phrase regenerates the service master key. When the service master key is regenerated, SQL Server decrypts all the keys that have been encrypted with it, and then encrypts them with the new service master key. This is a resource-intensive operation. You should schedule this operation during a period of low demand, unless the key has been compromised. If any one of the decryptions fail, the whole statement fails.

The FORCE option causes the key regeneration process to continue even if the process cannot retrieve the current master key, or cannot decrypt all the private keys that are encrypted with it. Use FORCE only if regeneration fails and you cannot restore the service master key by using the RESTORE SERVICE MASTER KEY statement.

Caution

The service master key is the root of the SQL Server encryption hierarchy. The service master key directly or indirectly protects all other keys and secrets in the tree. If a dependent key cannot be decrypted during a forced regeneration, the data the key secures will be lost.

The MACHINE KEY options allow you to add or drop encryption using the machine key.

Permissions

Requires CONTROL SERVER permission on the server.

Examples

The following example regenerates the service master key.

```
ALTER SERVICE MASTER KEY REGENERATE;
GO
```

See Also

RESTORE SERVICE MASTER KEY (Transact-SQL)

BACKUP SERVICE MASTER KEY (Transact-SQL)

Encryption Hierarchy

ALTER SYMMETRIC KEY

Changes the properties of a symmetric key.

 Transact-SQL Syntax Conventions

Syntax

ALTER SYMMETRIC KEY **Key_name** <alter_option>

<alter_option> ::=

ADD ENCRYPTION BY <encrypting_mechanism> [**,** ... **n**]

|

DROP ENCRYPTION BY <encrypting_mechanism> [**,** ... **n**]

<encrypting_mechanism> ::=

CERTIFICATE **certificate_name**

|

PASSWORD = '**password**'

|

SYMMETRIC KEY **Symmetric_Key_Name**

|

ASYMMETRIC KEY **Asym_Key_Name**

Arguments

Key_name

Is the name by which the symmetric key to be changed is known in the database.

ADD ENCRYPTION BY

Adds encryption by using the specified method.

DROP ENCRYPTION BY

Drops encryption by the specified method. You cannot remove all the encryptions from a symmetric key.

CERTIFICATE Certificate_name

Specifies the certificate that is used to encrypt the symmetric key. This certificate must already exist in the database.

PASSWORD = 'password'

Specifies the password that is used to encrypt the symmetric key. password must meet the Windows password policy requirements of the computer that is running the instance of SQL Server.

SYMMETRIC KEY Symmetric_Key_Name

Specifies the symmetric key that is used to encrypt the symmetric key that is being changed.
This symmetric key must already exist in the database and must be open.

ASYMMETRIC KEY Asym_Key_Name

Specifies the asymmetric key that is used to encrypt the symmetric key that is being changed.
This asymmetric key must already exist in the database.

Remarks

Caution

When a symmetric key is encrypted with a password instead of with the public key of the database master key, the TRIPLE_DES encryption algorithm is used. Because of this, keys that are created with a strong encryption algorithm, such as AES, are themselves secured by a weaker algorithm.

To change the encryption of the symmetric key, use the ADD ENCRYPTION and DROP ENCRYPTION phrases. It is never possible for a key to be entirely without encryption. For this reason, the best practice is to add the new form of encryption before removing the old form of encryption.

To change the owner of a symmetric key, use ALTER AUTHORIZATION.

Note

The RC4 algorithm is only supported for backward compatibility. New material can only be encrypted using RC4 or RC4_128 when the database is in compatibility level 90 or 100. (Not recommended.) Use a newer algorithm such as one of the AES algorithms instead. In SQL Server 2012 material encrypted using RC4 or RC4_128 can be decrypted in any compatibility level.

Permissions

Requires ALTER permission on the symmetric key. If adding encryption by a certificate or asymmetric key, requires VIEW DEFINITION permission on the certificate or asymmetric key. If dropping encryption by a certificate or asymmetric key, requires CONTROL permission on the certificate or asymmetric key.

Examples

The following example changes the encryption method that is used to protect a symmetric key. The symmetric key `JanainaKey043` is encrypted using certificate `Shipping04` when the key was created. Because the key can never be stored unencrypted, in this example, encryption is added by password, and then encryption is removed by certificate.

```
CREATE SYMMETRIC KEY JanainaKey043 WITH ALGORITHM = AES_256
    ENCRYPTION BY CERTIFICATE Shipping04;
-- Open the key.
OPEN SYMMETRIC KEY JanainaKey043 DECRYPTION BY CERTIFICATE Shipping04
```

```
    WITH PASSWORD = '<enterStrongPasswordHere>';
-- First, encrypt the key with a password.
ALTER SYMMETRIC KEY JanainaKey043
    ADD ENCRYPTION BY PASSWORD = '<enterStrongPasswordHere>';
-- Now remove encryption by the certificate.
ALTER SYMMETRIC KEY JanainaKey043
    DROP ENCRYPTION BY CERTIFICATE Shipping04;
CLOSE SYMMETRIC KEY JanainaKey043;
```

See Also

Encryption Hierarchy
OPEN SYMMETRIC KEY (Transact-SQL)
CLOSE SYMMETRIC KEY (Transact-SQL)
DROP SYMMETRIC KEY (Transact-SQL)
Encryption Hierarchy

ALTER TABLE

Modifies a table definition by altering, adding, or dropping columns and constraints, reassigning partitions, or disabling or enabling constraints and triggers.

 Transact-SQL Syntax Conventions

Syntax

```
ALTER TABLE [ database_name . [ schema_name ] . | schema_name . ] table_name
{
   ALTER COLUMN column_name
   {
      [ type_schema_name. ] type_name [ ( { precision [ , scale ]
         | max | xml_schema_collection } ) ]
      [ COLLATE collation_name ]
      [ NULL | NOT NULL ] [ SPARSE ]
   | {ADD | DROP }
      { ROWGUIDCOL | PERSISTED | NOT FOR REPLICATION | SPARSE }
   }
      | [ WITH { CHECK | NOCHECK } ]
```

```
| ADD
{
   <column_definition>
  | <computed_column_definition>
  | <table_constraint>
  | <column_set_definition>
} [ ,...n ]

| DROP
{
   [ CONSTRAINT ] constraint_name
   [ WITH ( <drop_clustered_constraint_option> [ ,...n ] ) ]
   | COLUMN column_name
} [ ,...n ]

| [ WITH { CHECK | NOCHECK } ] { CHECK | NOCHECK } CONSTRAINT
   { ALL | constraint_name [ ,...n ] }

| { ENABLE | DISABLE } TRIGGER
   { ALL | trigger_name [ ,...n ] }

| { ENABLE | DISABLE } CHANGE_TRACKING
   [ WITH ( TRACK_COLUMNS_UPDATED = { ON | OFF } ) ]

| SWITCH [ PARTITION source_partition_number_expression ]
   TO target_table
   [ PARTITION target_partition_number_expression ]

| SET ( FILESTREAM_ON = { partition_scheme_name | filegroup |
         "default" | "NULL" } )

| REBUILD
  [ [PARTITION = ALL]
   [ WITH ( <rebuild_option> [ ,...n ] ) ]
  | [ PARTITION = partition_number
```

```
        [ WITH ( <single_partition_rebuild_option> [ ,...n ] ) ]
      ]
    ]

  | (<table_option>)

  | (<filetable_option>)

}
[ ; ]



-- ALTER TABLE options


<column_set_definition> ::=
  column_set_name XML COLUMN_SET FOR ALL_SPARSE_COLUMNS

<drop_clustered_constraint_option> ::=
  {
    MAXDOP = max_degree_of_parallelism
      | ONLINE = {ON | OFF }
   | MOVE TO { partition_scheme_name ( column_name ) | filegroup
      | "default" }
  }
<table_option> ::=
  {
    SET ( LOCK_ESCALATION = { AUTO | TABLE | DISABLE } )
  }

<filetable_option> ::=
  {
    [ { ENABLE | DISABLE } FILETABLE_NAMESPACE ]
    [ SET ( FILETABLE_DIRECTORY = directory_name ) ]
```

}

<single_partition_rebuild__option> ::=

{

SORT_IN_TEMPDB = { ON | OFF }

| MAXDOP = **max_degree_of_parallelism**

| DATA_COMPRESSION = { NONE | ROW | PAGE} }

}

Arguments

database_name

Is the name of the database in which the table was created.

schema_name

Is the name of the schema to which the table belongs.

table_name

Is the name of the table to be altered. If the table is not in the current database or is not contained by the schema owned by the current user, the database and schema must be explicitly specified.

ALTER COLUMN

Specifies that the named column is to be changed or altered.

The modified column cannot be any one of the following:

- A column with a **timestamp** data type.
- The ROWGUIDCOL for the table.
- A computed column or used in a computed column.
- Used in an index, unless the column is a **varchar**, **nvarchar**, or **varbinary** data type, the data type is not changed, the new size is equal to or larger than the old size, and the index is not the result of a PRIMARY KEY constraint.
- Used in statistics generated by the CREATE STATISTICS statement unless the column is a **varchar**, **nvarchar**, or **varbinary** data type, the data type is not changed, and the new size is equal to or greater than the old size, or if the column is changed from not null to null. First, remove the statistics using the DROP STATISTICS statement. Statistics that are automatically generated by the query optimizer are automatically dropped by ALTER COLUMN.
- Used in a PRIMARY KEY or [FOREIGN KEY] REFERENCES constraint.
- Used in a CHECK or UNIQUE constraint. However, changing the length of a variable-length column used in a CHECK or UNIQUE constraint is allowed.
- Associated with a default definition. However, the length, precision, or scale of a column can be changed if the data type is not changed.

The data type of **text**, **ntext** and **image** columns can be changed only in the following ways:

- **text** to **varchar(max)**, **nvarchar(max)**, or **xml**
- **ntext** to **varchar(max)**, **nvarchar(max)**, or **xml**
- **image** to **varbinary(max)**

Some data type changes may cause a change in the data. For example, changing an **nchar** or **nvarchar** column to **char** or **varchar** may cause the conversion of extended characters. For more information, see CAST and CONVERT. Reducing the precision or scale of a column may cause data truncation.

The data type of a column of a partitioned table cannot be changed.

column_name

Is the name of the column to be altered, added, or dropped. column_name can be a maximum of 128 characters. For new columns, column_name can be omitted for columns created with a **timestamp** data type. The name **timestamp** is used if no column_name is specified for a **timestamp** data type column.

[type_schema_name.] type_name

Is the new data type for the altered column, or the data type for the added column. type_name cannot be specified for existing columns of partitioned tables. type_name can be any one of the following:

- A SQL Server system data type.
- An alias data type based on a SQL Server system data type. Alias data types are created with the CREATE TYPE statement before they can be used in a table definition.
- A .NET Framework user-defined type, and the schema to which it belongs. .NET Framework user-defined types are created with the CREATE TYPE statement before they can be used in a table definition.

The following are criteria for type_name of an altered column:

- The previous data type must be implicitly convertible to the new data type.
- type_name cannot be **timestamp**.
- ANSI_NULL defaults are always on for ALTER COLUMN; if not specified, the column is nullable.
- ANSI_PADDING padding is always ON for ALTER COLUMN.
- If the modified column is an identity column, new_data_type must be a data type that supports the identity property.
- The current setting for SET ARITHABORT is ignored. ALTER TABLE operates as if ARITHABORT is set to ON.

Note

If the COLLATE clause is not specified, changing the data type of a column will cause a collation

change to the default collation of the database.

precision

Is the precision for the specified data type. For more information about valid precision values, see Precision, Scale, and Length.

scale

Is the scale for the specified data type. For more information about valid scale values, see Precision, Scale, and Length.

max

Applies only to the **varchar**, **nvarchar**, and **varbinary** data types for storing 2^31-1 bytes of character, binary data, and of Unicode data.

xml_schema_collection

Applies only to the **xml** data type for associating an XML schema with the type. Before typing an **xml** column to a schema collection, the schema collection must first be created in the database by using CREATE XML SCHEMA COLLECTION.

COLLATE < collation_name >

Specifies the new collation for the altered column. If not specified, the column is assigned the default collation of the database. Collation name can be either a Windows collation name or a SQL collation name. For a list and more information, see Windows Collation Name and SQL Collation Name.

The COLLATE clause can be used to change the collations only of columns of the **char**, **varchar**, **nchar**, and **nvarchar** data types. To change the collation of a user-defined alias data type column, you must execute separate ALTER TABLE statements to change the column to a SQL Server system data type and change its collation, and then change the column back to an alias data type.

ALTER COLUMN cannot have a collation change if one or more of the following conditions exist:

- If a CHECK constraint, FOREIGN KEY constraint, or computed columns reference the column changed.
- If any index, statistics, or full-text index are created on the column. Statistics created automatically on the column changed are dropped if the column collation is changed.
- If a schema-bound view or function references the column.

For more information, see COLLATE.

NULL | NOT NULL

Specifies whether the column can accept null values. Columns that do not allow null values can be added with ALTER TABLE only if they have a default specified or if the table is empty. NOT NULL can be specified for computed columns only if PERSISTED is also specified. If the new column allows null values and no default is specified, the new column contains a null value for each row in the table. If the new column allows null values and a default definition is

added with the new column, WITH VALUES can be used to store the default value in the new column for each existing row in the table.

If the new column does not allow null values and the table is not empty, a DEFAULT definition must be added with the new column, and the new column automatically loads with the default value in the new columns in each existing row.

NULL can be specified in ALTER COLUMN to force a NOT NULL column to allow null values, except for columns in PRIMARY KEY constraints. NOT NULL can be specified in ALTER COLUMN only if the column contains no null values. The null values must be updated to some value before the ALTER COLUMN NOT NULL is allowed, for example:

```
UPDATE MyTable SET NullCol = N'some_value' WHERE NullCol IS
NULL;

ALTER TABLE MyTable ALTER COLUMN NullCOl NVARCHAR(20) NOT
NULL;
```

When you create or alter a table with the CREATE TABLE or ALTER TABLE statements, the database and session settings influence and possibly override the nullability of the data type that is used in a column definition. We recommend that you always explicitly define a column as NULL or NOT NULL for noncomputed columns.

If you add a column with a user-defined data type, we recommend that you define the column with the same nullability as the user-defined data type and specify a default value for the column. For more information, see CREATE TABLE.

Note

If NULL or NOT NULL is specified with ALTER COLUMN, new_data_type [(precision [, scale])] must also be specified. If the data type, precision, and scale are not changed, specify the current column values.

[{ADD | DROP} ROWGUIDCOL]

Specifies the ROWGUIDCOL property is added to or dropped from the specified column. ROWGUIDCOL indicates that the column is a row GUID column. Only one **uniqueidentifier** column per table can be designated as the ROWGUIDCOL column, and the ROWGUIDCOL property can be assigned only to a **uniqueidentifier** column. ROWGUIDCOL cannot be assigned to a column of a user-defined data type.

ROWGUIDCOL does not enforce uniqueness of the values that are stored in the column and does not automatically generate values for new rows that are inserted into the table. To generate unique values for each column, either use the NEWID function on INSERT statements or specify the NEWID function as the default for the column.

[{ADD | DROP} PERSISTED]

Specifies that the PERSISTED property is added to or dropped from the specified column. The column must be a computed column that is defined with a deterministic expression. For columns specified as PERSISTED, the Database Engine physically stores the computed values in the table and updates the values when any other columns on which the computed column depends are updated. By marking a computed column as PERSISTED, you can create indexes

on computed columns defined on expressions that are deterministic, but not precise. For more information, see Creating Indexes on Computed Columns.

Any computed column that is used as a partitioning column of a partitioned table must be explicitly marked PERSISTED.

DROP NOT FOR REPLICATION

Specifies that values are incremented in identity columns when replication agents perform insert operations. This clause can be specified only if column_name is an identity column.

SPARSE

Indicates that the column is a sparse column. The storage of sparse columns is optimized for null values. Sparse columns cannot be designated as NOT NULL. Converting a column from sparse to nonsparse or from nonsparse to sparse locks the table for the duration of the command execution. You may need to use the REBUILD clause to reclaim any space savings. For additional restrictions and more information about sparse columns, see Using Sparse Columns.

WITH CHECK | WITH NOCHECK

Specifies whether the data in the table is or is not validated against a newly added or re-enabled FOREIGN KEY or CHECK constraint. If not specified, WITH CHECK is assumed for new constraints, and WITH NOCHECK is assumed for re-enabled constraints.

If you do not want to verify new CHECK or FOREIGN KEY constraints against existing data, use WITH NOCHECK. We do not recommend doing this, except in rare cases. The new constraint will be evaluated in all later data updates. Any constraint violations that are suppressed by WITH NOCHECK when the constraint is added may cause future updates to fail if they update rows with data that does not comply with the constraint.

The query optimizer does not consider constraints that are defined WITH NOCHECK. Such constraints are ignored until they are re-enabled by using `ALTER TABLE table WITH CHECK CHECK CONSTRAINT ALL`.

ADD

Specifies that one or more column definitions, computed column definitions, or table constraints are added.

DROP { [CONSTRAINT] constraint_name | COLUMN column_name }

Specifies that constraint_name or column_name is removed from the table. Multiple columns and constraints can be listed.

The user-defined or system-supplied name of the constraint can be determined by querying the **sys.check_constraint**, **sys.default_constraints**, **sys.key_constraints**, and **sys.foreign_keys** catalog views.

A PRIMARY KEY constraint cannot be dropped if an XML index exists on the table.

A column cannot be dropped when it is:

- Used in an index.

- Used in a CHECK, FOREIGN KEY, UNIQUE, or PRIMARY KEY constraint.
- Associated with a default that is defined with the DEFAULT keyword, or bound to a default object.
- Bound to a rule.

Note

Dropping a column does not reclaim the disk space of the column. You may have to reclaim the disk space of a dropped column when the row size of a table is near, or has exceeded, its limit. Reclaim space by creating a clustered index on the table or rebuilding an existing clustered index by using ALTER INDEX.

WITH <drop_clustered_constraint_option>

Specifies that one or more drop clustered constraint options are set.

MAXDOP = max_degree_of_parallelism

Overrides the **max degree of parallelism** configuration option only for the duration of the operation. For more information, see Configure the max degree of parallelism Server Configuration Option.

Use the MAXDOP option to limit the number of processors used in parallel plan execution. The maximum is 64 processors.

max_degree_of_parallelism can be one of the following values:

1

Suppresses parallel plan generation.

>1

Restricts the maximum number of processors used in a parallel index operation to the specified number.

0 (default)

Uses the actual number of processors or fewer based on the current system workload.

For more information, see Configuring Parallel Index Operations.

Note

Parallel index operations are not available in every edition of SQL Server. For more information, see Features Supported by the Editions of SQL Server 2012.

ONLINE = { ON | OFF }

Specifies whether underlying tables and associated indexes are available for queries and data modification during the index operation. The default is OFF. REBUILD can be performed as an ONLINE operation.

ON

Long-term table locks are not held for the duration of the index operation. During the main phase of the index operation, only an Intent Share (IS) lock is held on the source

table. This enables queries or updates to the underlying table and indexes to continue. At the start of the operation, a Shared (S) lock is held on the source object for a very short time. At the end of the operation, for a short time, an S (Shared) lock is acquired on the source if a nonclustered index is being created; or an SCH-M (Schema Modification) lock is acquired when a clustered index is created or dropped online and when a clustered or nonclustered index is being rebuilt. ONLINE cannot be set to ON when an index is being created on a local temporary table. Only single-threaded heap rebuild operation is allowed.

OFF

Table locks are applied for the duration of the index operation. An offline index operation that creates, rebuilds, or drops a clustered index, or rebuilds or drops a nonclustered index, acquires a Schema modification (Sch-M) lock on the table. This prevents all user access to the underlying table for the duration of the operation. An offline index operation that creates a nonclustered index acquires a Shared (S) lock on the table. This prevents updates to the underlying table but allows read operations, such as SELECT statements. Multi-threaded heap rebuild operations are allowed.

For more information, see How Online Index Operations Work.

Note

Online index operations are not available in every edition of SQL Server. For more information, see Features Supported by the Editions of SQL Server 2012.

MOVE TO { partition_scheme_name (column_name [1, ... n]) | filegroup | "default" }

Specifies a location to move the data rows currently in the leaf level of the clustered index. The table is moved to the new location. This option applies only to constraints that create a clustered index.

Note

In this context, default is not a keyword. It is an identifier for the default filegroup and must be delimited, as in MOVE TO **"**default**"** or MOVE TO **[**default**]**. If **"**default**"** is specified, the QUOTED_IDENTIFIER option must be ON for the current session. This is the default setting. For more information, see SET QUOTED_IDENTIFIER (Transact-SQL).

{ CHECK | NOCHECK } CONSTRAINT

Specifies that constraint_name is enabled or disabled. This option can only be used with FOREIGN KEY and CHECK constraints. When NOCHECK is specified, the constraint is disabled and future inserts or updates to the column are not validated against the constraint conditions. DEFAULT, PRIMARY KEY, and UNIQUE constraints cannot be disabled.

ALL

Specifies that all constraints are either disabled with the NOCHECK option or enabled with the CHECK option.

{ ENABLE | DISABLE } TRIGGER

Specifies that trigger_name is enabled or disabled. When a trigger is disabled it is still defined for the table; however, when INSERT, UPDATE, or DELETE statements are executed against the table, the actions in the trigger are not performed until the trigger is re-enabled.

ALL

Specifies that all triggers in the table are enabled or disabled.

trigger_name

Specifies the name of the trigger to disable or enable.

{ ENABLE | DISABLE } CHANGE_TRACKING

Specifies whether change tracking is enabled disabled for the table. By default, change tracking is disabled.

This option is available only when change tracking is enabled for the database. For more information, see ALTER DATABASE SET Options (Transact-SQL).

To enable change tracking, the table must have a primary key.

WITH (TRACK_COLUMNS_UPDATED = { ON | OFF })

Specifies whether the Database Engine tracks which change tracked columns were updated. The default value is OFF.

SWITCH [PARTITION source_partition_number_expression] TO [schema_name.] target_table [PARTITION target_ partition_number_expression]

Switches a block of data in one of the following ways:

- Reassigns all data of a table as a partition to an already-existing partitioned table.
- Switches a partition from one partitioned table to another.
- Reassigns all data in one partition of a partitioned table to an existing non-partitioned table.

If table is a partitioned table, source_partition_number_expression must be specified. If target_table is partitioned, target_partition_number_expression must be specified. If reassigning a table's data as a partition to an already-existing partitioned table, or switching a partition from one partitioned table to another, the target partition must exist and it must be empty.

If reassigning one partition's data to form a single table, the target table must already be created and it must be empty. Both the source table or partition, and the target table or partition, must reside in the same filegroup. The corresponding indexes, or index partitions, must also reside in the same filegroup. Many additional restrictions apply to switching partitions. table and target_table cannot be the same. target_table can be a multi-part identifier.

source_partition_number_expression and target_partition_number_expression are constant expressions that can reference variables and functions. These include user-defined type variables and user-defined functions. They cannot reference Transact-SQL expressions.

For **SWITCH** restriction when using replication, see Replicate Partitioned Tables and Indexes.

SET (FILESTREAM_ON = { partition_scheme_name | filestream_filegroup_name | "default" | "NULL" })

Specifies where FILESTREAM data is stored.

ALTER TABLE with the SET FILESTREAM_ON clause will succeed only if the table has no FILESTREAM columns. The FILESTREAM columns can be added by using a second ALTER TABLE statement.

If partition_scheme_name is specified, the rules for CREATE TABLE apply. The table should already be partitioned for row data, and its partition scheme must use the same partition function and columns as the FILESTREAM partition scheme.

filestream_filegroup_name specifies the name of a FILESTREAM filegroup. The filegroup must have one file that is defined for the filegroup by using a CREATE DATABASE or ALTER DATABASE statement, or an error is raised.

"default**"** specifies the FILESTREAM filegroup with the DEFAULT property set. If there is no FILESTREAM filegroup, an error is raised.

"NULL**"** specifies that all references to FILESTREAM filegroups for the table will be removed. All FILESTREAM columns must be dropped first. You must use SET FILESTREAM_ON=**"**NULL**"** to delete all FILESTREAM data that is associated with a table.

SET (LOCK_ESCALATION = { AUTO | TABLE | DISABLE })

Specifies the allowed methods of lock escalation for a table.

AUTO

This option allows SQL Server Database Engine to select the lock escalation granularity that is appropriate for the table schema.

- If the table is partitioned, lock escalation will be allowed to partition. After the lock is escalated to the partition level, the lock will not be escalated later to TABLE granularity.
- If the table is not partitioned, the lock escalation will be done to the TABLE granularity.

TABLE

Lock escalation will be done at table-level granularity regardless whether the table is partitioned or not partitioned. This behavior is the same as in SQL Server 2005. TABLE is the default value.

DISABLE

Prevents lock escalation in most cases. Table-level locks are not completely disallowed. For example, when you are scanning a table that has no clustered index under the serializable isolation level, Database Engine must take a table lock to protect data integrity.

REBUILD

Use the REBUILD WITH syntax to rebuild an entire table including all the partitions in a

partitioned table. If the table has a clustered index, the REBUILD option rebuilds the clustered index. REBUILD can be performed as an ONLINE operation.

Use the REBUILD PARTITION syntax to rebuild a single partition in a partitioned table.

PARTITION = ALL

Rebuilds all partitions when changing the partition compression settings.

REBUILD WITH (<rebuild_option>)

All options apply to a table with a clustered index. If the table does not have a clustered index, the heap structure is only affected by some of the options.

When a specific compression setting is not specified with the REBUILD operation, the current compression setting for the partition is used. To return the current setting, query the **data_compression** column in the **sys.partitions** catalog view.

For complete descriptions of the rebuild options, see index_option (Transact-SQL).

DATA_COMPRESSION

Specifies the data compression option for the specified table, partition number, or range of partitions. The options are as follows:

NONE

Table or specified partitions are not compressed.

ROW

Table or specified partitions are compressed by using row compression.

PAGE

Table or specified partitions are compressed by using page compression.

To rebuild multiple partitions at the same time, see index_option (Transact-SQL). If the table does not have a clustered index, changing the data compression rebuilds the heap and the nonclustered indexes. For more information about compression, see Data Compression.

column_set_name XML COLUMN_SET FOR ALL_SPARSE_COLUMNS

Is the name of the column set. A column set is an untyped XML representation that combines all of the sparse columns of a table into a structured output. A column set cannot be added to a table that contains sparse columns. For more information about column sets, see Using Sparse Column Sets.

{ ENABLE | DISABLE } FILETABLE_NAMESPACE

Enables or disables the system-defined constraints on a FileTable. Can only be used with a FileTable.

SET (FILETABLE_DIRECTORY = directory_name)

Specifies the Windows-compatible FileTable directory name. This name should be unique among all the FileTable directory names in the database. Uniqueness comparison is case-

insensitive, regardless of SQL collation settings. Can only be used with a FileTable.

Remarks

To add new rows of data, use INSERT. To remove rows of data, use DELETE or TRUNCATE TABLE. To change the values in existing rows, use UPDATE.

If there are any execution plans in the procedure cache that reference the table, ALTER TABLE marks them to be recompiled on their next execution.

Changing the Size of a Column

You can change the length, precision, or scale of a column by specifying a new size for the column data type in the ALTER COLUMN clause. If data exists in the column, the new size cannot be smaller than the maximum size of the data. Also, the column cannot be defined in an index, unless the column is a **varchar**, **nvarchar**, or **varbinary** data type and the index is not the result of a PRIMARY KEY constraint. See example P.

Locks and ALTER TABLE

The changes specified in ALTER TABLE are implemented immediately. If the changes require modifications of the rows in the table, ALTER TABLE updates the rows. ALTER TABLE acquires a schema modify (SCH-M) lock on the table to make sure that no other connections reference even the metadata for the table during the change, except online index operations that require a very short SCH-M lock at the end. In an ALTER TABLE…SWITCH operation, the lock is acquired on both the source and target tables. The modifications made to the table are logged and fully recoverable. Changes that affect all the rows in very large tables, such as dropping a column or, on some editions of SQL Server, adding a NOT NULL column with a default value, can take a long time to complete and generate many log records. These ALTER TABLE statements should be executed with the same care as any INSERT, UPDATE, or DELETE statement that affects many rows.

Adding NOT NULL Columns as an Online Operation

In SQL Server 2012 Enterprise Edition, adding a NOT NULL column with a default value is an online operation when the default value is a *runtime constant*. This means that the operation is completed almost instantaneously regardless of the number of rows in the table. This is because the existing rows in the table are not updated during the operation; instead, the default value is stored only in the metadata of the table and the value is looked up as needed in queries that access these rows. This behavior is automatic; no additional syntax is required to implement the online operation beyond the ADD COLUMN syntax. A runtime constant is an expression that produces the same value at runtime for each row in the table regardless of its determinism. For example, the constant expression "My temporary data", or the system function GETUTCDATETIME() are runtime constants. In contrast, the functions NEWID() or NEWSEQUENTIALID() are not runtime constants because a unique value is produced for each row in the table. Adding a NOT NULL column with a default value that is not a runtime constant is always performed offline and an exclusive (SCH-M) lock is acquired for the duration of the operation.

While the existing rows reference the value stored in metadata, the default value is stored on the row for any new rows that are inserted and do not specify another value for the column. The default value stored in metadata is moved to an existing row when the row is updated (even if the actual column is not specified in the UPDATE statement), or if the table or clustered index is rebuilt.

Columns of type **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, **xml**, **text**, **ntext**, **image**, **hierarchyid**, **geometry**, **geography**, or CLR UDTS, cannot be added in an online operation. A column cannot be added online if doing so causes the maximum possible row size to exceed the 8,060 byte limit. The column is added as an offline operation in this case.

Parallel Plan Execution

In Microsoft SQL Server 2012 Enterprise, the number of processors employed to run a single ALTER TABLE ADD (index based) CONSTRAINT or DROP (clustered index) CONSTRAINT statement is determined by the **max degree of parallelism** configuration option and the current workload. If the Database Engine detects that the system is busy, the degree of parallelism of the operation is automatically reduced before statement execution starts. You can manually configure the number of processors that are used to run the statement by specifying the MAXDOP option. For more information, see Configure the max degree of parallelism Server Configuration Option.

Partitioned Tables

In addition to performing SWITCH operations that involve partitioned tables, ALTER TABLE can be used to change the state of the columns, constraints, and triggers of a partitioned table just like it is used for nonpartitioned tables. However, this statement cannot be used to change the way the table itself is partitioned. To repartition a partitioned table, use ALTER PARTITION SCHEME and ALTER PARTITION FUNCTION. Additionally, you cannot change the data type of a column of a partitioned table.

Restrictions on Tables with Schema-Bound Views

The restrictions that apply to ALTER TABLE statements on tables with schema-bound views are the same as the restrictions currently applied when modifying tables with a simple index. Adding a column is allowed. However, removing or changing a column that participates in any schema-bound view is not allowed. If the ALTER TABLE statement requires changing a column used in a schema-bound view, ALTER TABLE fails and the Database Engine raises an error message. For more information about schema binding and indexed views, see CREATE VIEW.

Adding or removing triggers on base tables is not affected by creating a schema-bound view that references the tables.

Indexes and ALTER TABLE

Indexes created as part of a constraint are dropped when the constraint is dropped. Indexes that were created with CREATE INDEX must be dropped with DROP INDEX. The ALTER INDEX statement can be used to rebuild an index part of a constraint definition; the constraint does not have to be dropped and added again with ALTER TABLE.

All indexes and constraints based on a column must be removed before the column can be removed.

When a constraint that created a clustered index is deleted, the data rows that were stored in the leaf level of the clustered index are stored in a nonclustered table. You can drop the clustered index and move the resulting table to another filegroup or partition scheme in a single transaction by specifying the MOVE TO option. The MOVE TO option has the following restrictions:

- MOVE TO is not valid for indexed views or nonclustered indexes.
- The partition scheme or filegroup must already exist.
- If MOVE TO is not specified, the table will be located in the same partition scheme or filegroup as was defined for the clustered index.

When you drop a clustered index, you can specify ONLINE = ON option so the DROP INDEX transaction does not block queries and modifications to the underlying data and associated nonclustered indexes.

ONLINE = ON has the following restrictions:

- ONLINE = ON is not valid for clustered indexes that are also disabled. Disabled indexes must be dropped by using ONLINE = OFF.
- Only one index at a time can be dropped.
- ONLINE = ON is not valid for indexed views, nonclustered indexes or indexes on local temp tables.

Temporary disk space equal to the size of the existing clustered index is required to drop a clustered index. This additional space is released as soon as the operation is completed.

Note
The options listed under <*drop_clustered_constraint_option*> apply to clustered indexes on tables and cannot be applied to clustered indexes on views or nonclustered indexes.

Replicating Schema Changes

By default, when you run ALTER TABLE on a published table at a SQL Server Publisher, that change is propagated to all SQL Server Subscribers. This functionality has some restrictions and can be disabled. For more information, see Making Schema Changes on Publication Databases.

Data Compression

System tables cannot be enabled for compression. . If the table is a heap, the rebuild operation for ONLINE mode will be single threaded. Use OFFLINE mode for a multi-threaded heap rebuild operation. For a more information about data compression, see Creating Compressed Tables and Indexes.

To evaluate how changing the compression state will affect a table, an index, or a partition, use the sp_estimate_data_compression_savings stored procedure.

The following restrictions apply to partitioned tables:

- You cannot change the compression setting of a single partition if the table has nonaligned indexes.
- The ALTER TABLE <table> REBUILD PARTITION ... syntax rebuilds the specified partition.
- The ALTER TABLE <table> REBUILD WITH ... syntax rebuilds all partitions.

Compatibility Support

The ALTER TABLE statement allows only two-part (schema.object) table names. In SQL Server 2012, specifying a table name using the following formats fails at compile time with error 117.

- server.database.schema.table
- .database.schema.table
- ..schema.table

In earlier versions specifying the format server.database.schema.table returned error 4902. Specifying the format .database.schema.table or the format ..schema.table succeeded.

To resolve the problem, remove the use of a 4-part prefix.

Permissions

Requires ALTER permission on the table.

ALTER TABLE permissions apply to both tables involved in an ALTER TABLE SWITCH statement. Any data that is switched inherits the security of the target table.

If any columns in the ALTER TABLE statement are defined to be of a common language runtime (CLR) user-defined type or alias data type, REFERENCES permission on the type is required.

Examples

A. Adding a new column

The following example adds a column that allows null values and has no values provided through a DEFAULT definition. In the new column, each row will have `NULL`.

```
CREATE TABLE dbo.doc_exa (column_a INT) ;
GO
ALTER TABLE dbo.doc_exa ADD column_b VARCHAR(20) NULL ;
GO
EXEC sp_help doc_exa ;
GO
DROP TABLE dbo.doc_exa ;
GO
```

B. Dropping a column

The following example modifies a table to remove a column.

```
CREATE TABLE dbo.doc_exb (column_a INT, column_b VARCHAR(20) NULL) ;
GO
```

```
ALTER TABLE dbo.doc_exb DROP COLUMN column_b ;
GO
EXEC sp_help doc_exb ;
GO
DROP TABLE dbo.doc_exb ;
GO
```

C. Changing the data type of a column

The following example changes a column of a table from `INT` to `DECIMAL`.

```
CREATE TABLE dbo.doc_exy (column_a INT ) ;
GO
INSERT INTO dbo.doc_exy (column_a) VALUES (10) ;
GO
ALTER TABLE dbo.doc_exy ALTER COLUMN column_a DECIMAL (5, 2) ;
GO
DROP TABLE dbo.doc_exy ;
GO
```

D. Adding a column with a constraint

The following example adds a new column with a `UNIQUE` constraint.

```
CREATE TABLE dbo.doc_exc (column_a INT) ;
GO
ALTER TABLE dbo.doc_exc ADD column_b VARCHAR(20) NULL
    CONSTRAINT exb_unique UNIQUE ;
GO
EXEC sp_help doc_exc ;
GO
DROP TABLE dbo.doc_exc ;
GO
```

E. Adding an unverified CHECK constraint to an existing column

The following example adds a constraint to an existing column in the table. The column has a value that violates the constraint. Therefore, `WITH NOCHECK` is used to prevent the constraint from being validated against existing rows, and to allow for the constraint to be added.

```
CREATE TABLE dbo.doc_exd ( column_a INT) ;
GO
INSERT INTO dbo.doc_exd VALUES (-1) ;
GO
```

```
ALTER TABLE dbo.doc_exd WITH NOCHECK
ADD CONSTRAINT exd_check CHECK (column_a > 1) ;
GO
EXEC sp_help doc_exd ;
GO
DROP TABLE dbo.doc_exd ;
GO
```

F. Adding a DEFAULT constraint to an existing column

The following example creates a table with two columns and inserts a value into the first column, and the other column remains NULL. A `DEFAULT` constraint is then added to the second column. To verify that the default is applied, another value is inserted into the first column, and the table is queried.

```
CREATE TABLE dbo.doc_exz ( column_a INT, column_b INT) ;
GO
INSERT INTO dbo.doc_exz (column_a)VALUES ( 7 ) ;
GO
ALTER TABLE dbo.doc_exz
ADD CONSTRAINT col_b_def
DEFAULT 50 FOR column_b ;
GO
INSERT INTO dbo.doc_exz (column_a) VALUES ( 10 ) ;
GO
SELECT * FROM dbo.doc_exz ;
GO
DROP TABLE dbo.doc_exz ;
GO
```

G. Adding several columns with constraints

The following example adds several columns with constraints defined with the new column. The first new column has an `IDENTITY` property. Each row in the table has new incremental values in the identity column.

```
CREATE TABLE dbo.doc_exe ( column_a INT CONSTRAINT column_a_un UNIQUE) ;
GO
ALTER TABLE dbo.doc_exe ADD

-- Add a PRIMARY KEY identity column.
column_b INT IDENTITY
```

```
CONSTRAINT column_b_pk PRIMARY KEY,

-- Add a column that references another column in the same table.
column_c INT NULL
CONSTRAINT column_c_fk
REFERENCES doc_exe(column_a),

-- Add a column with a constraint to enforce that
-- nonnull data is in a valid telephone number format.
column_d VARCHAR(16) NULL
CONSTRAINT column_d_chk
CHECK
(column_d LIKE '[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]' OR
column_d LIKE
'([0-9][0-9][0-9]) [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]'),

-- Add a nonnull column with a default.
column_e DECIMAL(3,3)
CONSTRAINT column_e_default
DEFAULT .081 ;
GO
EXEC sp_help doc_exe ;
GO
DROP TABLE dbo.doc_exe ;
GO
```

H. Adding a nullable column with default values

The following example adds a nullable column with a `DEFAULT` definition, and uses `WITH VALUES` to provide values for each existing row in the table. If WITH VALUES is not used, each row has the value NULL in the new column.

```
USE AdventureWorks2012 ;
GO
CREATE TABLE dbo.doc_exf ( column_a INT) ;
GO
INSERT INTO dbo.doc_exf VALUES (1) ;
GO
```

```
ALTER TABLE dbo.doc_exf
ADD AddDate smalldatetime NULL
CONSTRAINT AddDateDflt
DEFAULT GETDATE() WITH VALUES ;
GO
DROP TABLE dbo.doc_exf ;
GO
```

I. Disabling and re-enabling a constraint

The following example disables a constraint that limits the salaries accepted in the data. `NOCHECK CONSTRAINT` is used with `ALTER TABLE` to disable the constraint and allow for an insert that would typically violate the constraint. `CHECK CONSTRAINT` re-enables the constraint.

```
CREATE TABLE dbo.cnst_example
(id INT NOT NULL,
 name VARCHAR(10) NOT NULL,
 salary MONEY NOT NULL
    CONSTRAINT salary_cap CHECK (salary < 100000)
);

-- Valid inserts
INSERT INTO dbo.cnst_example VALUES (1,'Joe Brown',65000);
INSERT INTO dbo.cnst_example VALUES (2,'Mary Smith',75000);

-- This insert violates the constraint.
INSERT INTO dbo.cnst_example VALUES (3,'Pat Jones',105000);

-- Disable the constraint and try again.
ALTER TABLE dbo.cnst_example NOCHECK CONSTRAINT salary_cap;
INSERT INTO dbo.cnst_example VALUES (3,'Pat Jones',105000);

-- Re-enable the constraint and try another insert; this will fail.
ALTER TABLE dbo.cnst_example CHECK CONSTRAINT salary_cap;
INSERT INTO dbo.cnst_example VALUES (4,'Eric James',110000) ;
```

J. Dropping a constraint

The following example removes a `UNIQUE` constraint from a table.

```
CREATE TABLE dbo.doc_exc ( column_a INT
```

```
CONSTRAINT my_constraint UNIQUE) ;
GO
ALTER TABLE dbo.doc_exc DROP CONSTRAINT my_constraint ;
GO
DROP TABLE dbo.doc_exc ;
GO
```

K. Switching partitions between tables

The following example creates a partitioned table, assuming that partition scheme `myRangePS1` is already created in the database. Next, a non-partitioned table is created with the same structure as the partitioned table and on the same filegroup as `PARTITION 2` of table `PartitionTable`. The data of `PARTITION 2` of table `PartitionTable` is then switched into table `NonPartitionTable`.

```
CREATE TABLE PartitionTable (col1 int, col2 char(10))
ON myRangePS1 (col1) ;
GO
CREATE TABLE NonPartitionTable (col1 int, col2 char(10))
ON test2fg ;
GO
ALTER TABLE PartitionTable SWITCH PARTITION 2 TO NonPartitionTable ;
GO
```

L. Disabling and re-enabling a trigger

The following example uses the `DISABLE TRIGGER` option of `ALTER TABLE` to disable the trigger and allow for an insert that would typically violate the trigger. `ENABLE TRIGGER` is then used to re-enable the trigger.

```
CREATE TABLE dbo.trig_example
(id INT,
name VARCHAR(12),
salary MONEY) ;
GO
-- Create the trigger.
CREATE TRIGGER dbo.trig1 ON dbo.trig_example FOR INSERT
AS
IF (SELECT COUNT(*) FROM INSERTED
WHERE salary > 100000) > 0
BEGIN
```

```
    print 'TRIG1 Error: you attempted to insert a salary > $100,000'
    ROLLBACK TRANSACTION
END ;
GO
-- Try an insert that violates the trigger.
INSERT INTO dbo.trig_example VALUES (1,'Pat Smith',100001) ;
GO
-- Disable the trigger.
ALTER TABLE dbo.trig_example DISABLE TRIGGER trig1 ;
GO
-- Try an insert that would typically violate the trigger.
INSERT INTO dbo.trig_example VALUES (2,'Chuck Jones',100001) ;
GO
-- Re-enable the trigger.
ALTER TABLE dbo.trig_example ENABLE TRIGGER trig1 ;
GO
-- Try an insert that violates the trigger.
INSERT INTO dbo.trig_example VALUES (3,'Mary Booth',100001) ;
GO
```

M. Creating a PRIMARY KEY constraint with index options

The following example creates the PRIMARY KEY constraint `PK_TransactionHistoryArchive_TransactionID` and sets the options `FILLFACTOR`, `ONLINE`, and `PAD_INDEX`. The resulting clustered index will have the same name as the constraint.

```
USE AdventureWorks2012;
GO
ALTER TABLE Production.TransactionHistoryArchive WITH NOCHECK
ADD CONSTRAINT PK_TransactionHistoryArchive_TransactionID PRIMARY KEY
CLUSTERED (TransactionID)
WITH (FILLFACTOR = 75, ONLINE = ON, PAD_INDEX = ON);
GO
```

N. Dropping a PRIMARY KEY constraint in the ONLINE mode

The following example deletes a PRIMARY KEY constraint with the `ONLINE` option set to `ON`.

```
USE AdventureWorks2012;
GO
ALTER TABLE Production.TransactionHistoryArchive
```

```
DROP CONSTRAINT PK_TransactionHistoryArchive_TransactionID
WITH (ONLINE = ON);
GO
```

O. Adding and dropping a FOREIGN KEY constraint

The following example creates the table `ContactBackup`, and then alters the table, first by adding a `FOREIGN KEY` constraint that references the table `Person.Person`, then by dropping the `FOREIGN KEY` constraint.

```
USE AdventureWorks2012 ;
GO
CREATE TABLE Person.ContactBackup
(ContactID int) ;
GO
ALTER TABLE Person.ContactBackup
ADD CONSTRAINT FK_ContactBacup_Contact FOREIGN KEY (ContactID)
    REFERENCES Person.Person (BusinessEntityID) ;
ALTER TABLE Person.ContactBackup
DROP CONSTRAINT FK_ContactBacup_Contact ;
GO
DROP TABLE Person.ContactBackup ;
```

P. Changing the size of a column

The following example increases the size of a **varchar** column and the precision and scale of a **decimal** column. Because the columns contain data, the column size can only be increased. Also notice that `col_a` is defined in a unique index. The size of `col_a` can still be increased because the data type is a **varchar** and the index is not the result of a PRIMARY KEY constraint.

```
IF OBJECT_ID ( 'dbo.doc_exy', 'U' ) IS NOT NULL
    DROP TABLE dbo.doc_exy;
GO
-- Create a two-column table with a unique index on the varchar column.
CREATE TABLE dbo.doc_exy ( col_a varchar(5) UNIQUE NOT NULL, col_b decimal
(4,2));
GO
INSERT INTO dbo.doc_exy VALUES ('Test', 99.99);
GO
-- Verify the current column size.
SELECT name, TYPE_NAME(system_type_id), max_length, precision, scale
```

```
FROM sys.columns WHERE object_id = OBJECT_ID(N'dbo.doc_exy');
GO
-- Increase the size of the varchar column.
ALTER TABLE dbo.doc_exy ALTER COLUMN col_a varchar(25);
GO
-- Increase the scale and precision of the decimal column.
ALTER TABLE dbo.doc_exy ALTER COLUMN col_b decimal (10,4);
GO
-- Insert a new row.
INSERT INTO dbo.doc_exy VALUES ('MyNewColumnSize', 99999.9999) ;
GO
-- Verify the current column size.
SELECT name, TYPE_NAME(system_type_id), max_length, precision, scale
FROM sys.columns WHERE object_id = OBJECT_ID(N'dbo.doc_exy');
```

Q. Allowing lock escalation on partitioned tables

The following example enables lock escalation to the partition level on a partitioned table. If the table is not partitioned, lock escalation is to the TABLE level.

```
ALTER TABLE T1 SET (LOCK_ESCALATION = AUTO)
GO
```

R. Configuring change tracking on a table

The following example enables change tracking on the `Person.Person` table in the AdventureWorks2012 database.

```
USE AdventureWorks2012;
ALTER TABLE Person.Person
ENABLE CHANGE_TRACKING;
```

The following example enables change tracking and enables the tracking of the columns that are updated during a change.

```
USE AdventureWorks2012;
ALTER TABLE Person.Person
ENABLE CHANGE_TRACKING
WITH (TRACK_COLUMNS_UPDATED = ON)
```

The following example disables change tracking on the `Person.Person` table in the AdventureWorks2012 database:

```
USE AdventureWorks2012;
ALTER TABLE Person.Person
```

```
DISABLE CHANGE_TRACKING;
```

S. Modifying a table to change the compression

The following example changes the compression of a nonpartitioned table. The heap or clustered index will be rebuilt. If the table is a heap, all nonclustered indexes will be rebuilt.

```
ALTER TABLE T1
REBUILD WITH (DATA_COMPRESSION = PAGE);
```

The following example changes the compression of a partitioned table. The `REBUILD PARTITION = 1` syntax causes only partition number `1` to be rebuilt.

```
ALTER TABLE PartitionTable1
REBUILD PARTITION = 1 WITH (DATA_COMPRESSION =  NONE) ;
GO
```

The same operation using the following alternate syntax causes all partitions in the table to be rebuilt.

```
ALTER TABLE PartitionTable1
REBUILD PARTITION = ALL
WITH (DATA_COMPRESSION = PAGE ON PARTITIONS(1) ) ;
```

For additional data compression examples, see Creating Compressed Tables and Indexes.

T. Adding a sparse column

The following examples show adding and modifying sparse columns in table T1. The code to create table `T1` is as follows.

```
CREATE TABLE T1
(C1 int PRIMARY KEY,
C2 varchar(50) SPARSE NULL,
C3 int SPARSE NULL,
C4 int ) ;
GO
```

To add an additional sparse column `C5`, execute the following statement.

```
ALTER TABLE T1
ADD C5 char(100) SPARSE NULL ;
GO
```

To convert the `C4` non-sparse column to a sparse column, execute the following statement.

```
ALTER TABLE T1
ALTER COLUMN C4 ADD SPARSE ;
GO
```

To convert the C4 sparse column to a nonsparse column, execute the following statement.

```
ALTER TABLE T1
ALTER COLUMN C4 DROP SPARSE;
GO
```

U. Adding a column set

The following examples show adding a column to table T2. A column set cannot be added to a table that already contains sparse columns. The code to create table T2 is as follows.

```
CREATE TABLE T2
(C1 int PRIMARY KEY,
C2 varchar(50) NULL,
C3 int NULL,
C4 int ) ;
GO
```

The following three statements add a column set named CS, and then modify columns C2 and C3 to SPARSE.

```
ALTER TABLE T2
ADD CS XML COLUMN_SET FOR ALL_SPARSE_COLUMNS ;
GO

ALTER TABLE T2
ALTER COLUMN C2 ADD SPARSE ;
GO

ALTER TABLE T2
ALTER COLUMN C3 ADD SPARSE ;
GO
```

V. Changing column collation

The following example shows how to change the collation of a column. First we create table T3 with default user collations:

```
CREATE TABLE T3
(C1 int PRIMARY KEY,
C2 varchar(50) NULL,
C3 int NULL,
C4 int ) ;
```

```
GO
```

Next, column C2 collation is changed to Latin1_General_BIN. Note that the data type is required, even though it is not changed.

```
ALTER TABLE T3
ALTER COLUMN C2 varchar(50) COLLATE Latin1_General_BIN;
GO
```

See Also

sys.tables (Transact-SQL)
sp_rename
CREATE TABLE
DROP TABLE
sp_help
ALTER PARTITION SCHEME
ALTER PARTITION FUNCTION
EVENTDATA

column_definition

Specifies the properties of a column that are added to a table by using ALTER TABLE.

 Transact-SQL Syntax Conventions

Syntax

```
column_name [ type_schema_name. ] type_name
  [
        ( { precision [ , scale ] | max |
      [ { CONTENT | DOCUMENT } ] xml_schema_collection } )
  ]
  [ FILESTREAM ]
  [
    [ CONSTRAINT constraint_name ] DEFAULT constant_expression
        [ WITH VALUES ]
    | IDENTITY [ (seed , increment ) ] [ NOT FOR REPLICATION ]
  ]
  [ ROWGUIDCOL ]
  [ COLLATE < collation_name > ]
```

[<column_constraint> [...n]]

Arguments

column_name

Is the name of the column to be altered, added, or dropped. column_name can consist of 1 through 128 characters. For new columns, created with a timestamp data type, column_name can be omitted. If no column_name is specified for a **timestamp** data type column, the name **timestamp** is used.

[type_schema_name.] type_name

Is the data type for the column that is added and the schema to which it belongs.

type_name can be:

- A Microsoft SQL Server system data type.
- An alias data type based on a SQL Server system data type. Alias data types must be created by using CREATE TYPE before they can be used in a table definition.
- A Microsoft .NET Framework user-defined type and the schema to which it belongs. A .NET Framework user-defined type must be created by using CREATE TYPE before it can be used in a table definition.

If type_schema_name is not specified, the Microsoft Database Engine references type_name in the following order:

- The SQL Server system data type.
- The default schema of the current user in the current database.
- The **dbo** schema in the current database.

precision

Is the precision for the specified data type. For more information about valid precision values, see Precision, Scale, and Length.

scale

Is the scale for the specified data type. For more information about valid scale values, see Controlling Constraints, Identities, and Triggers with NOT FOR REPLICATION.

max

Applies only to the **varchar**, **nvarchar**, and **varbinary** data types. These are used for storing 2^31 bytes of character and binary data, and 2^30 bytes of Unicode data.

CONTENT

Specifies that each instance of the **xml** data type in column_name can comprise multiple top-level elements. CONTENT applies only to the **xml** data type and can be specified only if xml_schema_collection is also specified. If this is not specified, CONTENT is the default behavior.

DOCUMENT

Specifies that each instance of the **xml** data type in column_name can comprise only one

top-level element. DOCUMENT applies only to the **xml** data type and can be specified only if xml_schema_collection is also specified.

xml_schema_collection

Applies only to the **xml** data type for associating an XML schema collection with the type. Before typing an **xml** column to a schema, the schema must first be created in the database by using CREATE XML SCHEMA COLLECTION.

FILESTREAM

Optionally specifies the FILESTREAM storage attribute for column that has a type_name of **varbinary(max)**.

When FILESTREAM is specified for a column, the table must also have a column of the **uniqueidentifier** data type that has the ROWGUIDCOL attribute. This column must not allow null values and must have either a UNIQUE or PRIMARY KEY single-column constraint. The GUID value for the column must be supplied either by an application when data is being inserted, or by a DEFAULT constraint that uses the NEWID () function.

The ROWGUIDCOL column cannot be dropped and the related constraints cannot be changed while there is a FILESTREAM column defined for the table. The ROWGUIDCOL column can be dropped only after the last FILESTREAM column is dropped.

When the FILESTREAM storage attribute is specified for a column, all values for that column are stored in a FILESTREAM data container on the file system.

For an example that shows how to use column definition, see FILESTREAM (SQL Server).

[CONSTRAINT constraint_name]

Specifies the start of a DEFAULT definition. To maintain compatibility with earlier versions of SQL Server, a constraint name can be assigned to a DEFAULT. constraint_name must follow the rules for identifiers, except that the name cannot start with a number sign (#). If constraint_name is not specified, a system-generated name is assigned to the DEFAULT definition.

DEFAULT

Is a keyword that specifies the default value for the column. DEFAULT definitions can be used to provide values for a new column in the existing rows of data. DEFAULT definitions cannot be applied to **timestamp** columns, or columns with an IDENTITY property. If a default value is specified for a user-defined type column, the type should support an implicit conversion from constant_expression to the user-defined type.

constant_expression

Is a literal value, a NULL, or a system function used as the default column value. If used in conjunction with a column defined to be of a .NET Framework user-defined type, the implementation of the type must support an implicit conversion from the constant_expression to the user-defined type.

WITH VALUES

Specifies that the value given in DEFAULT constant_expression is stored in a new column added to existing rows. If the added column allows null values and WITH VALUES is specified, the default value is stored in the new column, added to existing rows. If WITH VALUES is not specified for columns that allow nulls, the value NULL is stored in the new column in existing rows. If the new column does not allow nulls, the default value is stored in new rows regardless of whether WITH VALUES is specified.

IDENTITY

Specifies that the new column is an identity column. The SQL Server Database Engine provides a unique, incremental value for the column. When you add identifier columns to existing tables, the identity numbers are added to the existing rows of the table with the seed and increment values. The order in which the rows are updated is not guaranteed. Identity numbers are also generated for any new rows that are added.

Identity columns are commonly used in conjunction with PRIMARY KEY constraints to serve as the unique row identifier for the table. The IDENTITY property can be assigned to a **tinyint**, **smallint**, **int**, **bigint**, **decimal(p,0)**, or **numeric(p,0)** column. Only one identity column can be created per table. The DEFAULT keyword and bound defaults cannot be used with an identity column. Either both the seed and increment must be specified, or neither. If neither are specified, the default is (1,1). You cannot modify an existing table column to add the IDENTITY property.

Note

Adding an identity column to a published table is not supported because it can result in nonconvergence when the column is replicated to the Subscriber. The values in the identity column at the Publisher depend on the order in which the rows for the affected table are physically stored. The rows might be stored differently at the Subscriber; therefore, the value for the identity column can be different for the same rows..

To disable the IDENTITY property of a column by allowing values to be explicitly inserted, use SET IDENTITY_INSERT.

seed

Is the value used for the first row loaded into the table.

increment

Is the incremental value added to the identity value of the previous row that is loaded.

NOT FOR REPLICATION

Can be specified for the IDENTITY property. If this clause is specified for the IDENTITY property, values are not incremented in identity columns when replication agents perform insert operations.

ROWGUIDCOL

Specifies that the column is a row globally unique identifier column. ROWGUIDCOL can only

be assigned to a **uniqueidentifier** column, and only one **uniqueidentifier** column per table can be designated as the ROWGUIDCOL column. ROWGUIDCOL cannot be assigned to columns of user-defined data types.

ROWGUIDCOL does not enforce uniqueness of the values stored in the column. Also, ROWGUIDCOL does not automatically generate values for new rows that are inserted into the table. To generate unique values for each column, either use the NEWID function on INSERT statements or specify the NEWID function as the default for the column. For more information, see NEWID (Transact-SQL)and INSERT (Transact-SQL).

COLLATE < collation_name >

Specifies the collation of the column. If not specified, the column is assigned the default collation of the database. Collation name can be either a Windows collation name or an SQL collation name. For a list and more information, see Windows Collation Name and SQL Collation Name.

The COLLATE clause can be used to specify the collations only of columns of the **char**, **varchar**, **nchar**, and **nvarchar** data types.

For more information about the COLLATE clause, see COLLATE.

Remarks

If a column is added having a **uniqueidentifier** data type, it can be defined with a default that uses the NEWID() function to supply the unique identifier values in the new column for each existing row in the table.

The Database Engine does not enforce an order for specifying DEFAULT, IDENTITY, ROWGUIDCOL, or column constraints in a column definition.

ALTER TABLE statement will fail if adding the column will cause the data row size to exceed 8060 bytes.

Examples

For examples, see ALTER TABLE (Transact-SQL).

See Also

ALTER TABLE

column_constraint

Specifies the properties of a PRIMARY KEY, FOREIGN KEY, UNIQUE, or CHECK constraint that is part of a new column definition added to a table by using ALTER TABLE.

Transact-SQL Syntax Conventions

Syntax

```
[ CONSTRAINT constraint_name ]
{
   [ NULL | NOT NULL ]
```

```
    { PRIMARY KEY | UNIQUE }
        [ CLUSTERED | NONCLUSTERED ]
        [ WITH FILLFACTOR = fillfactor ]
        [ WITH ( index_option [, ...n ] ) ]
        [ ON { partition_scheme_name (partition_column_name)
            | filegroup | "default" } ]
    | [ FOREIGN KEY ]
        REFERENCES [ schema_name . ] referenced_table_name
            [ ( ref_column ) ]
        [ ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
        [ ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
        [ NOT FOR REPLICATION ]
    | CHECK [ NOT FOR REPLICATION ] ( logical_expression )
}
```

Arguments

CONSTRAINT

Specifies the start of the definition for a PRIMARY KEY, UNIQUE, FOREIGN KEY, or CHECK constraint.

constraint_name

Is the name of the constraint. Constraint names must follow the rules for identifiers, except that the name cannot start with a number sign (#). If constraint_name is not supplied, a system-generated name is assigned to the constraint.

NULL | NOT NULL

Specifies whether the column can accept null values. Columns that do not allow null values can be added only if they have a default specified. If the new column allows null values and no default is specified, the new column contains NULL for each row in the table. If the new column allows null values and a default definition is added with the new column, the WITH VALUES option can be used to store the default value in the new column for each existing row in the table.

If the new column does not allow null values, a DEFAULT definition must be added with the new column. The new column automatically loads with the default value in the new columns in each existing row.

When the addition of a column requires physical changes to the data rows of a table, such as adding DEFAULT values to each row, locks are held on the table while ALTER TABLE runs. This affects the ability to change the content of the table while the lock is in place. In contrast, adding a column that allows null values and does not specify a default value is a metadata operation only, and involves no locks.

When you use CREATE TABLE or ALTER TABLE, database and session settings influence and possibly override the nullability of the data type that is used in a column definition. We recommend that you always explicitly define noncomputed columns as NULL or NOT NULL or, if you use a user-defined data type, that you allow the column to use the default nullability of the data type. For more information, see Controlling Constraints, Identities, and Triggers with NOT FOR REPLICATION.

PRIMARY KEY

Is a constraint that enforces entity integrity for a specified column or columns by using a unique index. Only one PRIMARY KEY constraint can be created for each table.

UNIQUE

Is a constraint that provides entity integrity for a specified column or columns by using a unique index.

CLUSTERED | NONCLUSTERED

Specifies that a clustered or nonclustered index is created for the PRIMARY KEY or UNIQUE constraint. PRIMARY KEY constraints default to CLUSTERED. UNIQUE constraints default to NONCLUSTERED.

If a clustered constraint or index already exists on a table, CLUSTERED cannot be specified. If a clustered constraint or index already exists on a table, PRIMARY KEY constraints default to NONCLUSTERED.

Columns that are of the **ntext**, **text**, **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, **xml**, or **image** data types cannot be specified as columns for an index.

WITH FILLFACTOR = fillfactor

Specifies how full the Database Engine should make each index page used to store the index data. User-specified fill factor values can be from 1 through 100. If a value is not specified, the default is 0.

Important

Documenting WITH FILLFACTOR = fillfactor as the only index option that applies to PRIMARY KEY or UNIQUE constraints is maintained for backward compatibility, but will not be documented in this manner in future releases. Other index options can be specified in the index_option clause of ALTER TABLE.

ON { partition_scheme_name (partition_column_name) | filegroup | "default" }

Specifies the storage location of the index created for the constraint. If partition_scheme_name is specified, the index is partitioned and the partitions are mapped to the filegroups that are specified by partition_scheme_name. If filegroup is specified, the index is created in the named filegroup. If "default" is specified or if ON is not specified at all, the index is created in the same filegroup as the table. If ON is specified when a clustered index is added for a PRIMARY KEY or UNIQUE constraint, the whole table is moved to the specified filegroup when the clustered index is created.

In this context, default, is not a keyword. It is an identifier for the default filegroup and must be delimited, as in ON "default" or ON [default]. If "default" is specified, the QUOTED_IDENTIFIER option must be ON for the current session. This is the default setting. For more information, see SET QUOTED_IDENTIFIER (Transact-SQL).

FOREIGN KEY REFERENCES

Is a constraint that provides referential integrity for the data in the column. FOREIGN KEY constraints require that each value in the column exist in the specified column in the referenced table.

schema_name

Is the name of the schema to which the table referenced by the FOREIGN KEY constraint belongs.

referenced_table_name

Is the table referenced by the FOREIGN KEY constraint.

ref_column

Is a column in parentheses referenced by the new FOREIGN KEY constraint.

ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT }

Specifies what action happens to rows in the table that is altered, if those rows have a referential relationship and the referenced row is deleted from the parent table. The default is NO ACTION.

NO ACTION

The SQL Server Database Engine raises an error and the delete action on the row in the parent table is rolled back.

CASCADE

Corresponding rows are deleted from the referencing table if that row is deleted from the parent table.

SET NULL

All the values that make up the foreign key are set to NULL when the corresponding row in the parent table is deleted. For this constraint to execute, the foreign key columns must be nullable.

SET DEFAULT

All the values that comprise the foreign key are set to their default values when the corresponding row in the parent table is deleted. For this constraint to execute, all foreign key columns must have default definitions. If a column is nullable and there is no explicit default value set, NULL becomes the implicit default value of the column.

Do not specify CASCADE if the table will be included in a merge publication that uses logical records. For more information about logical records, see Grouping Changes to Related Rows with Logical Records.

The ON DELETE CASCADE cannot be defined if an INSTEAD OF trigger ON DELETE already exists on the table that is being altered.

For example, in the database, the **ProductVendor** table has a referential relationship with the **Vendor** table. The **ProductVendor.VendorID** foreign key references the **Vendor.VendorID** primary key.

If a DELETE statement is executed on a row in the **Vendor** table, and an ON DELETE CASCADE action is specified for **ProductVendor.VendorID**, the Database Engine checks for one or more dependent rows in the **ProductVendor** table. If any exist, the dependent rows in the **ProductVendor** table will be deleted, in addition to the row referenced in the **Vendor** table.

Conversely, if NO ACTION is specified, the Database Engine raises an error and rolls back the delete action on the **Vendor** row when there is at least one row in the **ProductVendor** table that references it.

ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT }

Specifies what action happens to rows in the table altered when those rows have a referential relationship and the referenced row is updated in the parent table. The default is NO ACTION.

NO ACTION

The Database Engine raises an error, and the update action on the row in the parent table is rolled back.

CASCADE

Corresponding rows are updated in the referencing table when that row is updated in the parent table.

SET NULL

All the values that make up the foreign key are set to NULL when the corresponding row in the parent table is updated. For this constraint to execute, the foreign key columns must be nullable.

SET DEFAULT

All the values that make up the foreign key are set to their default values when the corresponding row in the parent table is updated. For this constraint to execute, all foreign key columns must have default definitions. If a column is nullable and there is no explicit default value set, NULL becomes the implicit default value of the column.

Do not specify CASCADE if the table will be included in a merge publication that uses logical records. For more information about logical records, see Grouping Changes to Related Rows with Logical Records.

ON UPDATE CASCADE, SET NULL, or SET DEFAULT cannot be defined if an INSTEAD OF trigger ON UPDATE already exists on the table that is being altered.

For example, in the database, the **ProductVendor** table has a referential relationship with the **Vendor** table. The **ProductVendor.VendorID** foreign key references the

Vendor.VendorID primary key.

If an UPDATE statement is executed on a row in the **Vendor** table and an ON UPDATE CASCADE action is specified for **ProductVendor.VendorID**, the Database Engine checks for one or more dependent rows in the **ProductVendor** table. If any exist, the dependent row in the **ProductVendor** table will be updated, in addition to the row referenced in the **Vendor** table.

Conversely, if NO ACTION is specified, the Database Engine raises an error and rolls back the update action on the **Vendor** row when there is at least one row in the **ProductVendor** table that references it.

NOT FOR REPLICATION

Can be specified for FOREIGN KEY constraints and CHECK constraints. If this clause is specified for a constraint, the constraint is not enforced when replication agents perform insert, update, or delete operations.

CHECK

Is a constraint that enforces domain integrity by limiting the possible values that can be entered into a column or columns.

logical_expression

Is a logical expression used in a CHECK constraint and returns TRUE or FALSE. logical_expression used with CHECK constraints cannot reference another table but can reference other columns in the same table for the same row. The expression cannot reference an alias data type.

Remarks

When FOREIGN KEY or CHECK constraints are added, all existing data is verified for constraint violations unless the WITH NOCHECK option is specified. If any violations occur, ALTER TABLE fails and an error is returned. When a new PRIMARY KEY or UNIQUE constraint is added to an existing column, the data in the column or columns must be unique. If duplicate values are found, ALTER TABLE fails. The WITH NOCHECK option has no effect when PRIMARY KEY or UNIQUE constraints are added.

Each PRIMARY KEY and UNIQUE constraint generates an index. The number of UNIQUE and PRIMARY KEY constraints cannot cause the number of indexes on the table to exceed 999 nonclustered indexes and 1 clustered index. Foreign key constraints do not automatically generate an index. However, foreign key columns are frequently used in join criteria in queries by matching the column or columns in the foreign key constraint of one table with the primary or unique key column or columns in the other table. An index on the foreign key columns enables the Database Engine to quickly find related data in the foreign key table.

Examples

For examples, see ALTER TABLE (Transact-SQL).

See Also

ALTER TABLE

column_definition

computed_column_definition

Specifies the properties of a computed column that is added to a table by using ALTER TABLE.

 Transact-SQL Syntax Conventions

Syntax

```
column_name AS computed_column_expression
[ PERSISTED [ NOT NULL ] ]
[
   [ CONSTRAINT constraint_name ]
   { PRIMARY KEY | UNIQUE }
      [ CLUSTERED | NONCLUSTERED ]
      [ WITH FILLFACTOR = fillfactor ]
      [ WITH ( <index_option> [ , ...n ] ) ]
      [ ON { partition_scheme_name ( partition_column_name ) | filegroup
         | "default" } ]
   | [ FOREIGN KEY ]
      REFERENCES ref_table [ ( ref_column ) ]
      [ ON DELETE { NO ACTION | CASCADE } ]
      [ ON UPDATE { NO ACTION } ]
      [ NOT FOR REPLICATION ]
   | CHECK [ NOT FOR REPLICATION ] ( logical_expression )
]
```

Arguments

column_name

Is the name of the column to be altered, added, or dropped. column_name can be 1 through 128 characters. For new columns, column_name can be omitted for columns created with a **timestamp** data type. If no column_name is specified for a **timestamp** data type column, the name **timestamp** is used.

computed_column_expression

Is an expression that defines the value of a computed column. A computed column is a virtual column that is not physically stored in the table but is computed from an expression that uses other columns in the same table. For example, a computed column could have the definition: cost AS price * qty. The expression can be a noncomputed column name, constant, function, variable, and any combination of these connected by one or more operators. The

expression cannot be a subquery or include an alias data type.

Computed columns can be used in select lists, WHERE clauses, ORDER BY clauses, or any other locations where regular expressions can be used, but with the following exceptions:

- A computed column cannot be used as a DEFAULT or FOREIGN KEY constraint definition or with a NOT NULL constraint definition. However, if the computed column value is defined by a deterministic expression and the data type of the result is allowed in index columns, a computed column can be used as a key column in an index or as part of any PRIMARY KEY or UNIQUE constraint.

  For example, if the table has integer columns a and b, the computed column a + b may be indexed, but computed column a + DATEPART(dd, GETDATE()) cannot be indexed, because the value might change in subsequent invocations.

- A computed column cannot be the target of an INSERT or UPDATE statement.

Note

Because each row in a table can have different values for columns involved in a computed column, the computed column may not have the same result for each row.

PERSISTED

Specifies that the Database Engine will physically store the computed values in the table, and update the values when any other columns on which the computed column depends are updated. Marking a computed column as PERSISTED allows an index to be created on a computed column that is deterministic, but not precise. For more information, see Controlling Constraints, Identities, and Triggers with NOT FOR REPLICATION. Any computed columns used as partitioning columns of a partitioned table must be explicitly marked PERSISTED. computed_column_expression must be deterministic when PERSISTED is specified.

NULL | NOT NULL

Specifies whether null values are allowed in the column. NULL is not strictly a constraint but can be specified like NOT NULL. NOT NULL can be specified for computed columns only if PERSISTED is also specified.

CONSTRAINT

Specifies the start of the definition for a PRIMARY KEY or UNIQUE constraint.

constraint_name

Is the new constraint. Constraint names must follow the rules for identifiers, except that the name cannot start with a number sign (#). If constraint_name is not supplied, a system-generated name is assigned to the constraint.

PRIMARY KEY

Is a constraint that enforces entity integrity for a specified column or columns by using a unique index. Only one PRIMARY KEY constraint can be created for each table.

UNIQUE

Is a constraint that provides entity integrity for a specific column or columns by using a unique index.

CLUSTERED | NONCLUSTERED

Specifies that a clustered or nonclustered index is created for the PRIMARY KEY or UNIQUE constraint. PRIMARY KEY constraints default to CLUSTERED. UNIQUE constraints default to NONCLUSTERED.

If a clustered constraint or index already exists on a table, CLUSTERED cannot be specified. If a clustered constraint or index already exists on a table, PRIMARY KEY constraints default to NONCLUSTERED.

WITH FILLFACTOR = fillfactor

Specifies how full the SQL Server Database Engine should make each index page used to store the index data. User-specified fillfactor values can be from 1 through 100. If a value is not specified, the default is 0.

Important

Documenting WITH FILLFACTOR = fillfactor as the only index option that applies to PRIMARY KEY or UNIQUE constraints is maintained for backward compatibility, but will not be documented in this manner in future releases. Other index options can be specified in the index_option clause of ALTER TABLE.

FOREIGN KEY REFERENCES

Is a constraint that provides referential integrity for the data in the column or columns. FOREIGN KEY constraints require that each value in the column exists in the corresponding referenced column or columns in the referenced table. FOREIGN KEY constraints can reference only columns that are PRIMARY KEY or UNIQUE constraints in the referenced table or columns referenced in a UNIQUE INDEX on the referenced table. Foreign keys on computed columns must also be marked PERSISTED.

ref_table

Is the name of the table referenced by the FOREIGN KEY constraint.

(ref_column)

Is a column from the table referenced by the FOREIGN KEY constraint.

ON DELETE { NO ACTION | CASCADE }

Specifies what action happens to rows in the table if those rows have a referential relationship and the referenced row is deleted from the parent table. The default is NO ACTION.

NO ACTION

The Database Engine raises an error and the delete action on the row in the parent table is rolled back.

CASCADE

Corresponding rows are deleted from the referencing table if that row is deleted from the parent table.

For example, in the database, the ProductVendor table has a referential relationship with the Vendor table. The ProductVendor.BusinessEntityID foreign key references the Vendor.BusinessEntityID primary key.

If a DELETE statement is executed on a row in the Vendor table, and an ON DELETE CASCADE action is specified for ProductVendor.BusinessEntityID, the Database Engine checks for one or more dependent rows in the ProductVendor table. If any exist, the dependent rows in the ProductVendor table are deleted, in addition to the row referenced in the Vendor table.

Conversely, if NO ACTION is specified, the Database Engine raises an error and rolls back the delete action on the Vendor row when there is at least one row in the ProductVendor table that references it.

Do not specify CASCADE if the table will be included in a merge publication that uses logical records. For more information about logical records, see Grouping Changes to Related Rows with Logical Records.

ON UPDATE { NO ACTION }

Specifies what action happens to rows in the table created when those rows have a referential relationship and the referenced row is updated in the parent table. When NO ACTION is specified, the Database Engine raises an error and rolls back the update action on the Vendor row if there is at least one row in the ProductVendor table that references it.

NOT FOR REPLICATION

Can be specified for FOREIGN KEY constraints and CHECK constraints. If this clause is specified for a constraint, the constraint is not enforced when replication agents perform insert, update, or delete operations.

CHECK

Is a constraint that enforces domain integrity by limiting the possible values that can be entered into a column or columns. CHECK constraints on computed columns must also be marked PERSISTED.

logical_expression

Is a logical expression that returns TRUE or FALSE. The expression cannot contain a reference to an alias data type.

ON { partition_scheme_name (partition_column_name) | filegroup| "default"}

Specifies the storage location of the index created for the constraint. If partition_scheme_name is specified, the index is partitioned and the partitions are mapped to the filegroups that are specified by partition_scheme_name. If filegroup is specified, the index is created in the named filegroup. If "default" is specified or if ON is not specified at all, the index is created in the same filegroup as the table. If ON is specified when a clustered index is

added for a PRIMARY KEY or UNIQUE constraint, the whole table is moved to the specified filegroup when the clustered index is created.

Note

In this context, default is not a keyword. It is an identifier for the default filegroup and must be delimited, as in ON "default" or ON [default]. If "default" is specified, the QUOTED_IDENTIFIER option must be ON for the current session. This is the default setting. For more information, see SET QUOTED_IDENTIFIER (Transact-SQL).

Remarks

Each PRIMARY KEY and UNIQUE constraint generates an index. The number of UNIQUE and PRIMARY KEY constraints cannot cause the number of indexes on the table to exceed 999 nonclustered indexes and 1 clustered index.

See Also

ALTER TABLE

table_constraint

Specifies the properties of a PRIMARY KEY, UNIQUE, FOREIGN KEY, or CHECK constraint, or a DEFAULT definition added to a table by using ALTER TABLE.

Transact-SQL Syntax Conventions

Syntax

```
[ CONSTRAINT constraint_name ]
{
   { PRIMARY KEY | UNIQUE }
      [ CLUSTERED | NONCLUSTERED ]
      (column [ ASC | DESC ] [ ,...n ] )
      [ WITH FILLFACTOR = fillfactor
      [ WITH ( <index_option>[ , ...n ] ) ]
      [ ON { partition_scheme_name ( partition_column_name ... )
        | filegroup | "default" } ]
   | FOREIGN KEY
      ( column [ ,...n ] )
      REFERENCES referenced_table_name [ ( ref_column [ ,...n ] ) ]
      [ ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
      [ ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
      [ NOT FOR REPLICATION ]
   | DEFAULT constant_expression FOR column [ WITH VALUES ]
```

| CHECK [NOT FOR REPLICATION] **(`logical_expression`)**

}

Arguments

CONSTRAINT

Specifies the start of a definition for a PRIMARY KEY, UNIQUE, FOREIGN KEY, or CHECK constraint, or a DEFAULT.

constraint_name

Is the name of the constraint. Constraint names must follow the rules for identifiers, except that the name cannot start with a number sign (#). If constraint_name is not supplied, a system-generated name is assigned to the constraint.

PRIMARY KEY

Is a constraint that enforces entity integrity for a specified column or columns by using a unique index. Only one PRIMARY KEY constraint can be created for each table.

UNIQUE

Is a constraint that provides entity integrity for a specified column or columns by using a unique index.

CLUSTERED | NONCLUSTERED

Specifies that a clustered or nonclustered index is created for the PRIMARY KEY or UNIQUE constraint. PRIMARY KEY constraints default to CLUSTERED. UNIQUE constraints default to NONCLUSTERED.

If a clustered constraint or index already exists on a table, CLUSTERED cannot be specified. If a clustered constraint or index already exists on a table, PRIMARY KEY constraints default to NONCLUSTERED.

Columns that are of the **ntext**, **text**, **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, **xml**, or **image** data types cannot be specified as columns for an index.

column

Is a column or list of columns specified in parentheses that are used in a new constraint.

[ASC | DESC]

Specifies the order in which the column or columns participating in table constraints are sorted. The default is ASC.

WITH FILLFACTOR = fillfactor

Specifies how full the Database Engine should make each index page used to store the index data. User-specified fillfactor values can be from 1 through 100. If a value is not specified, the default is 0.

Important

Documenting WITH FILLFACTOR = fillfactor as the only index option that applies to PRIMARY KEY or

UNIQUE constraints is maintained for backward compatibility, but will not be documented in this manner in future releases. Other index options can be specified in the index_option clause of ALTER TABLE.

ON { partition_scheme_name (partition_column_name) | filegroup | "default" }

Specifies the storage location of the index created for the constraint. If partition_scheme_name is specified, the index is partitioned and the partitions are mapped to the filegroups that are specified by partition_scheme_name. If filegroup is specified, the index is created in the named filegroup. If "default" is specified or if ON is not specified at all, the index is created in the same filegroup as the table. If ON is specified when a clustered index is added for a PRIMARY KEY or UNIQUE constraint, the whole table is moved to the specified filegroup when the clustered index is created.

In this context, default is not a keyword; it is an identifier for the default filegroup and must be delimited, as in ON "default" or ON [default]. If "default" is specified, the QUOTED_IDENTIFIER option must be ON for the current session. This is the default setting.

FOREIGN KEY REFERENCES

Is a constraint that provides referential integrity for the data in the column. FOREIGN KEY constraints require that each value in the column exist in the specified column in the referenced table.

referenced_table_name

Is the table referenced by the FOREIGN KEY constraint.

ref_column

Is a column or list of columns in parentheses referenced by the new FOREIGN KEY constraint.

ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT }

Specifies what action happens to rows in the table that is altered, if those rows have a referential relationship and the referenced row is deleted from the parent table. The default is NO ACTION.

NO ACTION

The SQL Server Database Engine raises an error and the delete action on the row in the parent table is rolled back.

CASCADE

Corresponding rows are deleted from the referencing table if that row is deleted from the parent table.

SET NULL

All the values that make up the foreign key are set to NULL when the corresponding row in the parent table is deleted. For this constraint to execute, the foreign key columns must be nullable.

SET DEFAULT

All the values that comprise the foreign key are set to their default values when the corresponding row in the parent table is deleted. For this constraint to execute, all foreign key columns must have default definitions. If a column is nullable and there is no explicit default value set, NULL becomes the implicit default value of the column.

Do not specify CASCADE if the table will be included in a merge publication that uses logical records. For more information about logical records, see Grouping Changes to Related Rows with Logical Records.

ON DELETE CASCADE cannot be defined if an INSTEAD OF trigger ON DELETE already exists on the table that is being altered.

For example, in the database, the **ProductVendor** table has a referential relationship with the **Vendor** table. The **ProductVendor.VendorID** foreign key references the **Vendor.VendorID** primary key.

If a DELETE statement is executed on a row in the **Vendor** table and an ON DELETE CASCADE action is specified for **ProductVendor.VendorID**, the Database Engine checks for one or more dependent rows in the **ProductVendor** table. If any exist, the dependent rows in the **ProductVendor** table will be deleted, in addition to the row referenced in the **Vendor** table.

Conversely, if NO ACTION is specified, the Database Engine raises an error and rolls back the delete action on the **Vendor** row when there is at least one row in the **ProductVendor** table that references it.

ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT }

Specifies what action happens to rows in the table altered when those rows have a referential relationship and the referenced row is updated in the parent table. The default is NO ACTION.

NO ACTION

The Database Engine raises an error, and the update action on the row in the parent table is rolled back.

CASCADE

Corresponding rows are updated in the referencing table when that row is updated in the parent table.

SET NULL

All the values that make up the foreign key are set to NULL when the corresponding row in the parent table is updated. For this constraint to execute, the foreign key columns must be nullable.

SET DEFAULT

All the values that make up the foreign key are set to their default values when the corresponding row in the parent table is updated. For this constraint to execute, all foreign key columns must have default definitions. If a column is nullable, and there is no explicit default value set, NULL becomes the implicit default value of the column.

Do not specify CASCADE if the table will be included in a merge publication that uses logical records. For more information about logical records, see Grouping Changes to Related Rows with Logical Records.

ON UPDATE CASCADE, SET NULL, or SET DEFAULT cannot be defined if an INSTEAD OF trigger ON UPDATE already exists on the table that is being altered.

For example, in the database, the **ProductVendor** table has a referential relationship with the **Vendor** table. The **ProductVendor.VendorID** foreign key references the **Vendor.VendorID** primary key.

If an UPDATE statement is executed on a row in the **Vendor** table and an ON UPDATE CASCADE action is specified for **ProductVendor.VendorID**, the Database Engine checks for one or more dependent rows in the **ProductVendor** table. If any exist, the dependent row in the **ProductVendor** table will be updated, as well as the row referenced in the **Vendor** table.

Conversely, if NO ACTION is specified, the Database Engine raises an error and rolls back the update action on the **Vendor** row when there is at least one row in the **ProductVendor** table that references it.

NOT FOR REPLICATION

Can be specified for FOREIGN KEY constraints and CHECK constraints. If this clause is specified for a constraint, the constraint is not enforced when replication agents perform insert, update, or delete operations.

DEFAULT

Specifies the default value for the column. DEFAULT definitions can be used to provide values for a new column in the existing rows of data. DEFAULT definitions cannot be added to columns that have a **timestamp** data type, an IDENTITY property, an existing DEFAULT definition, or a bound default. If the column has an existing default, the default must be dropped before the new default can be added. If a default value is specified for a user-defined type column, the type should support an implicit conversion from constant_expression to the user-defined type. To maintain compatibility with earlier versions of SQL Server, a constraint name can be assigned to a DEFAULT.

constant_expression

Is a literal value, a NULL, or a system function that is used as the default column value. If constant_expression is used in conjunction with a column defined to be of a Microsoft .NET Framework user-defined type, the implementation of the type must support an implicit conversion from the constant_expression to the user-defined type.

FOR column

Specifies the column associated with a table-level DEFAULT definition.

WITH VALUES

Specifies that the value given in DEFAULT constant_expression is stored in a new column that is added to existing rows. WITH VALUES can be specified only when DEFAULT is specified in an ADD column clause. If the added column allows null values and WITH VALUES is specified,

the default value is stored in the new column that is added to existing rows. If WITH VALUES is not specified for columns that allow nulls, NULL is stored in the new column in existing rows. If the new column does not allow nulls, the default value is stored in new rows regardless of whether WITH VALUES is specified.

CHECK

Is a constraint that enforces domain integrity by limiting the possible values that can be entered into a column or columns.

logical_expression

Is a logical expression used in a CHECK constraint and returns TRUE or FALSE. logical_expression used with CHECK constraints cannot reference another table but can reference other columns in the same table for the same row. The expression cannot reference an alias data type.

Remarks

When FOREIGN KEY or CHECK constraints are added, all existing data is verified for constraint violations unless the WITH NOCHECK option is specified. If any violations occur, ALTER TABLE fails and an error is returned. When a new PRIMARY KEY or UNIQUE constraint is added to an existing column, the data in the column or columns must be unique. If duplicate values are found, ALTER TABLE fails. The WITH NOCHECK option has no effect when PRIMARY KEY or UNIQUE constraints are added.

Each PRIMARY KEY and UNIQUE constraint generates an index. The number of UNIQUE and PRIMARY KEY constraints cannot cause the number of indexes on the table to exceed 999 nonclustered indexes and 1 clustered index. Foreign key constraints do not automatically generate an index. However, foreign key columns are frequently used in join criteria in queries by matching the column or columns in the foreign key constraint of one table with the primary or unique key column or columns in the other table. An index on the foreign key columns enables the Database Engine to quickly find related data in the foreign key table.

Examples

For examples, see ALTER TABLE (Transact-SQL).

See Also

ALTER TABLE

index_option

Specifies a set of options that can be applied to an index that is part of a constraint definition that is created by using ALTER TABLE.

Transact-SQL Syntax Conventions

Syntax

```
{
```

```
    PAD_INDEX = { ON | OFF }
  | FILLFACTOR = fillfactor
  | IGNORE_DUP_KEY = { ON | OFF }
  | STATISTICS_NORECOMPUTE = { ON | OFF }
  | ALLOW_ROW_LOCKS = { ON | OFF }
  | ALLOW_PAGE_LOCKS = { ON | OFF }
  | SORT_IN_TEMPDB = { ON | OFF }
  | ONLINE = { ON | OFF }
  | MAXDOP = max_degree_of_parallelism
  | DATA_COMPRESSION = { NONE |ROW | PAGE}
     [ ON PARTITIONS ( { <partition_number_expression> | <range> }
     [ , ...n ] ) ]
}

<range> ::=
<partition_number_expression> TO <partition_number_expression>

<single_partition_rebuild__option> ::=
{
    SORT_IN_TEMPDB = { ON | OFF }
  | MAXDOP = max_degree_of_parallelism
  | DATA_COMPRESSION = {NONE | ROW | PAGE } }
}
```

Arguments

PAD_INDEX = { ON | OFF }

Specifies index padding. The default is OFF.

ON

The percentage of free space that is specified by FILLFACTOR is applied to the intermediate-level pages of the index.

OFF or fillfactor is not specified

The intermediate-level pages are filled to near capacity, leaving enough space for at least one row of the maximum size the index can have, given the set of keys on the intermediate pages.

FILLFACTOR = fillfactor

Specifies a percentage that indicates how full the Database Engine should make the leaf level of each index page during index creation or alteration. The value specified must be an

integer value from 1 to 100. The default is 0.

Note

Fill factor values 0 and 100 are identical in all respects.

IGNORE_DUP_KEY = { ON | OFF }

Specifies the error response when an insert operation attempts to insert duplicate key values into a unique index. The IGNORE_DUP_KEY option applies only to insert operations after the index is created or rebuilt. The option has no effect when executing CREATE INDEX, ALTER INDEX, or UPDATE. The default is OFF.

ON

A warning message will occur when duplicate key values are inserted into a unique index. Only the rows violating the uniqueness constraint will fail.

OFF

An error message will occur when duplicate key values are inserted into a unique index. The entire INSERT operation will be rolled back.

IGNORE_DUP_KEY cannot be set to ON for indexes created on a view, non-unique indexes, XML indexes, spatial indexes, and filtered indexes.

To view IGNORE_DUP_KEY, use sys.indexes.

In backward compatible syntax, WITH IGNORE_DUP_KEY is equivalent to WITH IGNORE_DUP_KEY = ON.

STATISTICS_NORECOMPUTE = { ON | OFF }

Specifies whether statistics are recomputed. The default is OFF.

ON

Out-of-date statistics are not automatically recomputed.

OFF

Automatic statistics updating are enabled.

ALLOW_ROW_LOCKS = { ON | OFF }

Specifies whether row locks are allowed. The default is ON.

ON

Row locks are allowed when accessing the index. The Database Engine determines when row locks are used.

OFF

Row locks are not used.

ALLOW_PAGE_LOCKS = { ON | OFF }

Specifies whether page locks are allowed. The default is ON.

ON

Page locks are allowed when accessing the index. The Database Engine determines when page locks are used.

OFF

Page locks are not used.

SORT_IN_TEMPDB = { ON | OFF }

Specifies whether to store sort results in **tempdb**. The default is OFF.

ON

The intermediate sort results that are used to build the index are stored in **tempdb**. This may reduce the time required to create an index if **tempdb** is on a different set of disks than the user database. However, this increases the amount of disk space that is used during the index build.

OFF

The intermediate sort results are stored in the same database as the index.

ONLINE = { ON | OFF }

Specifies whether underlying tables and associated indexes are available for queries and data modification during the index operation. The default is OFF.

Note

Unique nonclustered indexes cannot be created online. This includes indexes that are created due to a UNIQUE or PRIMARY KEY constraint.

ON

Long-term table locks are not held for the duration of the index operation. During the main phase of the index operation, only an Intent Share (IS) lock is held on the source table. This enables queries or updates to the underlying table and indexes to proceed. At the start of the operation, a Shared (S) lock is held on the source object for a very short period of time. At the end of the operation, for a short period of time, an S (Shared) lock is acquired on the source if a nonclustered index is being created; or an SCH-M (Schema Modification) lock is acquired when a clustered index is created or dropped online and when a clustered or nonclustered index is being rebuilt. ONLINE cannot be set to ON when an index is being created on a local temporary table.

OFF

Table locks are applied for the duration of the index operation. An offline index operation that creates, rebuilds, or drops a clustered index, or rebuilds or drops a nonclustered index, acquires a Schema modification (Sch-M) lock on the table. This prevents all user access to the underlying table for the duration of the operation. An offline index operation that creates a nonclustered index acquires a Shared (S) lock on the table. This prevents updates to the underlying table but allows read operations, such as SELECT statements.

For more information, see How Online Index Operations Work.

Note

Online index operations are not available in every edition of Microsoft SQL Server. For a list of features that are supported by the editions of SQL Server, see Features Supported by the Editions of SQL Server 2012.

MAXDOP = max_degree_of_parallelism

Overrides the **max degree of parallelism** configuration option for the duration of the index operation. For more information, see Configure the max degree of parallelism Server Configuration Option. Use MAXDOP to limit the number of processors used in a parallel plan execution. The maximum is 64 processors.

max_degree_of_parallelism can be:

1

Suppresses parallel plan generation.

>1

Restricts the maximum number of processors used in a parallel index operation to the specified number.

0 (default)

Uses the actual number of processors or fewer based on the current system workload.

For more information, see Configuring Parallel Index Operations.

Note

Parallel index operations are not available in every edition of Microsoft SQL Server. For a list of features that are supported by the editions of SQL Server, see Features Supported by the Editions of SQL Server 2012.

DATA_COMPRESSION

Specifies the data compression option for the specified table, partition number or range of partitions. The options are as follows:

NONE

Table or specified partitions are not compressed.

ROW

Table or specified partitions are compressed by using row compression.

PAGE

Table or specified partitions are compressed by using page compression.

For more information about compression, see Creating Compressed Tables and Indexes.

ON PARTITIONS ({ <partition_number_expression> | <range> } [,...n])

Specifies the partitions to which the DATA_COMPRESSION setting applies. If the table is not

partitioned, the ON PARTITIONS argument will generate an error. If the ON PARTITIONS clause is not provided, the DATA_COMPRESSION option will apply to all partitions of a partitioned table.

<partition_number_expression> can be specified in the following ways:

- Provide the number a partition, for example: ON PARTITIONS (2).
- Provide the partition numbers for several individual partitions separated by commas, for example: ON PARTITIONS (1, 5).
- Provide both ranges and individual partitions, for example: ON PARTITIONS (2, 4, 6 TO 8).

<range> can be specified as partition numbers separated by the word TO, for example: ON PARTITIONS (6 TO 8).

To set different types of data compression for different partitions, specify the DATA_COMPRESSION option more than once, for example:

```
REBUILD WITH
(
DATA_COMPRESSION = NONE ON PARTITIONS (1),
DATA_COMPRESSION = ROW ON PARTITIONS (2, 4, 6 TO 8),
DATA_COMPRESSION = PAGE ON PARTITIONS (3, 5)
)
```

<single_partition_rebuild__option>

In most cases, rebuilding an index rebuilds all partitions of a partitioned index. The following options, when applied to a single partition, do not rebuild all partitions.

- SORT_IN_TEMPDB
- MAXDOP
- DATA_COMPRESSION

Remarks

For a complete description of index options, see CREATE INDEX (Transact-SQL).

See Also

ALTER TABLE
column_constraint
computed_column_definition
table_constraint

ALTER TRIGGER

Modifies the definition of a DML, DDL, or logon trigger that was previously created by the CREATE TRIGGER statement. Triggers are created by using CREATE TRIGGER. They can be

created directly from Transact-SQL statements or from methods of assemblies that are created in the Microsoft .NET Framework common language runtime (CLR) and uploaded to an instance of SQL Server. For more information about the parameters that are used in the ALTER TRIGGER statement, see Making Schema Changes on Publication Databases.

 Transact-SQL Syntax Conventions

Syntax

```
Trigger on an INSTEAD OF INSERT, UPDATE, or DELETE statement to a table or view (DML Trigger)
ALTER TRIGGER schema_name.trigger_name
ON ( table | view )
[ WITH <dml_trigger_option> [ ,...n ] ]
( FOR | AFTER | INSTEAD OF )
{ [ DELETE ] [ , ] [ INSERT ] [ , ] [ UPDATE ] }
[ NOT FOR REPLICATION ]
AS { sql_statement [ ; ] [ ...n ] | EXTERNAL NAME <method specifier> [ ; ] }

<dml_trigger_option> ::=
    [ ENCRYPTION ]
    [ <EXECUTE AS Clause> ]

<method_specifier> ::=
    assembly_name.class_name.method_name

Trigger on a CREATE, ALTER, DROP, GRANT, DENY, REVOKE, or UPDATE statement (DDL Trigger)

ALTER TRIGGER trigger_name
ON { DATABASE | ALL SERVER }
[ WITH <ddl_trigger_option> [ ,...n ] ]
{ FOR | AFTER } { event_type [ ,...n ] | event_group }
AS { sql_statement [ ; ] | EXTERNAL NAME <method specifier>
[ ; ] }
}

<ddl_trigger_option> ::=
```

[ENCRYPTION]
[<**EXECUTE AS Clause**>]

<method_specifier> ::=
assembly_name.class_name.method_name

Trigger on a LOGON event (Logon Trigger)
ALTER TRIGGER **trigger_name**
ON ALL SERVER
[WITH <logon_trigger_option> [**,...n**]]
{ FOR | AFTER } LOGON
AS { **sql_statement** [;] [**,...n**] | EXTERNAL NAME < method specifier > [;] }

<logon_trigger_option> ::=
[ENCRYPTION]
[**EXECUTE AS Clause**]

<method_specifier> ::=
assembly_name.class_name.method_name

Arguments

schema_name

Is the name of the schema to which a DML trigger belongs. DML triggers are scoped to the schema of the table or view on which they are created. schema_name is optional only if the DML trigger and its corresponding table or view belong to the default schema. schema_name cannot be specified for DDL or logon triggers.

trigger_name

Is the existing trigger to modify.

table | view

Is the table or view on which the DML trigger is executed. Specifying the fully-qualified name of the table or view is optional.

DATABASE

Applies the scope of a DDL trigger to the current database. If specified, the trigger fires whenever event_type or event_group occurs in the current database.

ALL SERVER

Applies the scope of a DDL or logon trigger to the current server. If specified, the trigger fires

whenever event_type or event_group occurs anywhere in the current server.

WITH ENCRYPTION

Encrypts the sys.syscomments sys.sql_modules entries that contain the text of the ALTER TRIGGER statement. Using WITH ENCRYPTION prevents the trigger from being published as part of SQL Server replication. WITH ENCRYPTION cannot be specified for CLR triggers.

Note

If a trigger is created by using WITH ENCRYPTION, it must be specified again in the ALTER TRIGGER statement for this option to remain enabled.

EXECUTE AS

Specifies the security context under which the trigger is executed. Enables you to control the user account the instance of SQL Server uses to validate permissions on any database objects that are referenced by the trigger.

For more information, see EXECUTE AS.

AFTER

Specifies that the trigger is fired only after the triggering SQL statement is executed successfully. All referential cascade actions and constraint checks also must have been successful before this trigger fires.

AFTER is the default, if only the FOR keyword is specified.

DML AFTER triggers may be defined only on tables.

INSTEAD OF

Specifies that the DML trigger is executed instead of the triggering SQL statement, therefore, overriding the actions of the triggering statements. INSTEAD OF cannot be specified for DDL or logon triggers.

At most, one INSTEAD OF trigger per INSERT, UPDATE, or DELETE statement can be defined on a table or view. However, you can define views on views where each view has its own INSTEAD OF trigger.

INSTEAD OF triggers are not allowed on views created by using WITH CHECK OPTION. SQL Server raises an error when an INSTEAD OF trigger is added to a view for which WITH CHECK OPTION was specified. The user must remove that option using ALTER VIEW before defining the INSTEAD OF trigger.

{ [DELETE] [,] [INSERT] [,] [UPDATE] } | { [INSERT] [,] [UPDATE] }

Specifies the data modification statements, when tried against this table or view, activate the DML trigger. At least one option must be specified. Any combination of these in any order is allowed in the trigger definition. If more than one option is specified, separate the options with commas.

For INSTEAD OF triggers, the DELETE option is not allowed on tables that have a referential relationship specifying a cascade action ON DELETE. Similarly, the UPDATE option is not allowed on tables that have a referential relationship specifying a cascade action ON

UPDATE. For more information, see ALTER TABLE (Transact-SQL).

event_type

Is the name of a Transact-SQL language event that, after execution, causes a DDL trigger to fire. Valid events for DDL triggers are listed in DDL Events.

event_group

Is the name of a predefined grouping of Transact-SQL language events. The DDL trigger fires after execution of any Transact-SQL language event that belongs to event_group. Valid event groups for DDL triggers are listed in DDL Event Groups. After ALTER TRIGGER has finished running, event_group also acts as a macro by adding the event types it covers to the sys.trigger_events catalog view.

NOT FOR REPLICATION

Indicates that the trigger should not be executed when a replication agent modifies the table involved in the trigger.

sql_statement

Is the trigger conditions and actions.

<method_specifier>

Specifies the method of an assembly to bind with the trigger. The method must take no arguments and return void. class_name must be a valid SQL Server identifier and must exist as a class in the assembly with assembly visibility. The class cannot be a nested class.

Remarks

For more information about ALTER TRIGGER, see Remarks in CREATE TRIGGER.

Note
The EXTERNAL_NAME and ON_ALL_SERVER options are not available in a contained database.

DML Triggers

ALTER TRIGGER supports manually updatable views through INSTEAD OF triggers on tables and views. SQL Server applies ALTER TRIGGER the same way for all kinds of triggers (AFTER, INSTEAD-OF).

The first and last AFTER triggers to be executed on a table can be specified by using sp_settriggerorder. Only one first and one last AFTER trigger can be specified on a table. If there are other AFTER triggers on the same table, they are randomly executed.

If an ALTER TRIGGER statement changes a first or last trigger, the first or last attribute set on the modified trigger is dropped, and the order value must be reset by using sp_settriggerorder.

An AFTER trigger is executed only after the triggering SQL statement has executed successfully. This successful execution includes all referential cascade actions and constraint checks associated with the object updated or deleted. The AFTER trigger operation checks for the

effects of the triggering statement and also all referential cascade UPDATE and DELETE actions that are caused by the triggering statement.

When a DELETE action to a child or referencing table is the result of a CASCADE on a DELETE from the parent table, and an INSTEAD OF trigger on DELETE is defined on that child table, the trigger is ignored and the DELETE action is executed.

DDL Triggers

Unlike DML triggers, DDL triggers are not scoped to schemas. Therefore, the OBJECT_ID, OBJECT_NAME, OBJECTPROPERTY, and OBJECTPROPERTY(EX) cannot be used when querying metadata about DDL triggers. Use the catalog views instead. For more information, see Getting Information About DDL Triggers.

Permissions

To alter a DML trigger requires ALTER permission on the table or view on which the trigger is defined.

To alter a DDL trigger defined with server scope (ON ALL SERVER) or a logon trigger requires CONTROL SERVER permission on the server. To alter a DDL trigger defined with database scope (ON DATABASE) requires ALTER ANY DATABASE DDL TRIGGER permission in the current database.

Examples

The following example creates a DML trigger that prints a user-defined message to the client when a user tries to add or change data in the `SalesPersonQuotaHistory` table. The trigger is then modified by using `ALTER TRIGGER` to apply the trigger only on `INSERT` activities. This trigger is helpful because it reminds the user that updates or inserts rows into this table to also notify the `Compensation` department.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID(N'Sales.bonus_reminder', N'TR') IS NOT NULL
    DROP TRIGGER Sales.bonus_reminder;
GO
CREATE TRIGGER Sales.bonus_reminder
ON Sales.SalesPersonQuotaHistory
WITH ENCRYPTION
AFTER INSERT, UPDATE
AS RAISERROR ('Notify Compensation', 16, 10);
GO
-- Now, change the trigger.
USE AdventureWorks2012;
GO
```

```
ALTER TRIGGER Sales.bonus_reminder
ON Sales.SalesPersonQuotaHistory
AFTER INSERT
AS RAISERROR ('Notify Compensation', 16, 10);
GO
```

See Also

DROP TRIGGER
ENABLE TRIGGER
DISABLE TRIGGER
EVENTDATA
sp_helptrigger
Create a Stored Procedure
sp_addmessage (Transact-SQL)
Transactions
Getting Information About DML Triggers
Getting Information about DDL Triggers
sys.triggers
sys.trigger_events
sys.sql_modules
sys.assembly_modules
sys.server_triggers
sys.server_trigger_events
sys.server_sql_modules
sys.server_assembly_modules
Making Schema Changes on Publication Databases

ALTER USER

Renames a database user or changes its default schema.

 Transact-SQL Syntax Conventions

Syntax

ALTER USER **userName**
    WITH <set_item> [**,**...n]

<set_item> ::=

NAME = **newUserName**

| DEFAULT_SCHEMA = { **schemaName** | NULL }

| LOGIN = **loginName**

| **PASSWORD = 'password' [OLD_PASSWORD = 'oldpassword'**]

| **DEFAULT_LANGUAGE** = { NONE | <lcid> | <language name> | <language alias> }

Arguments

userName

Specifies the name by which the user is identified inside this database.

LOGIN = loginName

Re-maps a user to another login by changing the user's Security Identifier (SID) to match the login's SID.

NAME = newUserName

Specifies the new name for this user. newUserName must not already occur in the current database.

DEFAULT_SCHEMA = { schemaName | NULL }

Specifies the first schema that will be searched by the server when it resolves the names of objects for this user. Setting the default schema to NULL removes a default schema from a Windows group. The NULL option cannot be used with a Windows user.

PASSWORD = 'password'

Specifies the password for the user that is being changed. Passwords are case-sensitive.

Note

This option is available only for contained users. See Understanding Contained Databases and sp_migrate_user_to_contained for more information.

OLD_PASSWORD = 'oldpassword'

The current user password that will be replaced by 'password'. Passwords are case-sensitive. OLD_PASSWORD is required to change a password, unless you have **ALTER ANY USER** permission. Requiring OLD_PASSWORD prevents users with **IMPERSONATION** permission from changing the password.

Note

This option is available only for contained users.

DEFAULT_LANGUAGE = { NONE | <lcid> | <language name> | <language alias> }

Specifies a default language to be assigned to the user. If this option is set to NONE, the default language is set to the current default language of the database. If the default

language of the database is later changed, the default language of the user will remain unchanged. DEFAULT_LANGUAGE can be the local ID (lcid), the name of the language, or the language alias.

Note

This option may only be specified in a contained database and only for contained users.

Remarks

The default schema will be the first schema that will be searched by the server when it resolves the names of objects for this database user. Unless otherwise specified, the default schema will be the owner of objects created by this database user.

If the user has a default schema, that default schema will used. If the user does not have a default schema, but the user is a member of a group that has a default schema, the default schema of the group will be used. If the user does not have a default schema, and is a member of more than one group that has a default schema, the schema of the Windows group with the lowest **principle_id** will be used. If no default schema can be determined for a user, the **dbo** schema will be used.

DEFAULT_SCHEMA can be set to a schema that does not currently occur in the database. Therefore, you can assign a DEFAULT_SCHEMA to a user before that schema is created.

DEFAULT_SCHEMA cannot be specified for a user who is mapped to a certificate, or an asymmetric key.

Important

The value of DEFAULT_SCHEMA is ignored if the user is a member of the **sysadmin** fixed server role. All members of the **sysadmin** fixed server role have a default schema of `dbo`.

You can change the name of a user who is mapped to a Windows login or group only when the SID of the new user name matches the SID that is recorded in the database. This check helps prevent spoofing of Windows logins in the database.

The WITH LOGIN clause enables the remapping of a user to a different login. Users without a login, users mapped to a certificate, or users mapped to an asymmetric key cannot be re-mapped with this clause. Only SQL users and Windows users (or groups) can be remapped. The WITH LOGIN clause cannot be used to change the type of user, such as changing a Windows account to a SQL Server login.

The name of the user will be automatically renamed to the login name if the following conditions are true.

- The user is a Windows user.
- The name is a Windows name (contains a backslash).
- No new name was specified.
- The current name differs from the login name.

Otherwise, the user will not be renamed unless the caller additionally invokes the NAME clause.

The name of a user mapped to a SQL Server login, a certificate, or an asymmetric key cannot contain the backslash character (\).

Caution

Beginning with SQL Server 2005, the behavior of schemas changed. As a result, code that assumes that schemas are equivalent to database users may no longer return correct results. Old catalog views, including , should not be used in a database in which any of the following DDL statements have ever been used: CREATE SCHEMA, ALTER SCHEMA, DROP SCHEMA, CREATE USER, ALTER USER, DROP USER, CREATE ROLE, ALTER ROLE, DROP ROLE, CREATE APPROLE, ALTER APPROLE, DROP APPROLE, ALTER AUTHORIZATION. In such databases you must instead use the new catalog views. The new catalog views take into account the separation of principals and schemas that was introduced in SQL Server 2005. For more information about catalog views, see .

Security

Note

A user who has **ALTER ANY USER** permission can change the default schema of any user. A user who has an altered schema might unknowingly select data from the wrong table or execute code from the wrong schema.

Permissions

To change the name of a user or remap the user to a different login requires the **ALTER ANY USER** permission.

To change the default schema or language requires **ALTER** permission on the user. Users can change only their own default schema or language.

Examples

A. Changing the name of a database user

The following example changes the name of the database user `Mary5` to `Mary51`.

```
USE AdventureWorks2012;

ALTER USER Mary5 WITH NAME = Mary51;

GO
```

B. Changing the default schema of a user

The following example changes the default schema of the user `Mary51` to `Purchasing`.

```
USE AdventureWorks2012;

ALTER USER Mary51 WITH DEFAULT_SCHEMA = Purchasing;

GO
```

C. Changing several options at once

The following example changes several options for a contained database user in one statement.

```
USE AdventureWorks2012;
GO
ALTER USER Philip
WITH  NAME = Philipe
    , DEFAULT_SCHEMA = Development
    , PASSWORD = 'W1r77TT98%ab@#' OLD_PASSWORD = 'New Devel0per'
    , DEFAULT_LANGUAGE  = French ;
GO
```

See Also

CREATE USER (Transact-SQL)
DROP USER (Transact-SQL)
Understanding Contained Databases
eventdata (Transact-SQL)
sp_migrate_user_to_contained

ALTER VIEW

Modifies a previously created view. This includes an indexed view. ALTER VIEW does not affect dependent stored procedures or triggers and does not change permissions.

 Transact-SQL Syntax Conventions

Syntax

ALTER VIEW [**schema_name** .] **view_name** [**(** **column** [**,**...**n**] **)**]
[WITH <view_attribute> [**,**...n]]
AS **select_statement**
[WITH CHECK OPTION] [;]

<view_attribute> ::=
{
   [ENCRYPTION]
   [SCHEMABINDING]
   [VIEW_METADATA]
}

Arguments

schema_name

Is the name of the schema to which the view belongs.

view_name

Is the view to change.

column

Is the name of one or more columns, separated by commas, that are to be part of the specified view.

Important

Column permissions are maintained only when columns have the same name before and after ALTER VIEW is performed.

Note

In the columns for the view, the permissions for a column name apply across a CREATE VIEW or ALTER VIEW statement, regardless of the source of the underlying data. For example, if permissions are granted on the **SalesOrderID** column in a CREATE VIEW statement, an ALTER VIEW statement can rename the **SalesOrderID** column, such as to **OrderRef**, and still have the permissions associated with the view using **SalesOrderID**.

ENCRYPTION

Encrypts the entries in sys.syscomments that contain the text of the ALTER VIEW statement. WITH ENCRYPTION prevents the view from being published as part of SQL Server replication.

SCHEMABINDING

Binds the view to the schema of the underlying table or tables. When SCHEMABINDING is specified, the base tables cannot be modified in a way that would affect the view definition. The view definition itself must first be modified or dropped to remove dependencies on the table to be modified. When you use SCHEMABINDING, the select_statement must include the two-part names (schema**.**object) of tables, views, or user-defined functions that are referenced. All referenced objects must be in the same database.

Views or tables that participate in a view created with the SCHEMABINDING clause cannot be dropped, unless that view is dropped or changed so that it no longer has schema binding. Otherwise, the Database Engine raises an error. Also, executing ALTER TABLE statements on tables that participate in views that have schema binding fail if these statements affect the view definition.

VIEW_METADATA

Specifies that the instance of SQL Server will return to the DB-Library, ODBC, and OLE DB APIs the metadata information about the view, instead of the base table or tables, when browse-mode metadata is being requested for a query that references the view. Browse-mode metadata is additional metadata that the instance of Database Engine returns to the client-side DB-Library, ODBC, and OLE DB APIs. This metadata enables the client-side APIs to

implement updatable client-side cursors. Browse-mode metadata includes information about the base table that the columns in the result set belong to.

For views created with VIEW_METADATA, the browse-mode metadata returns the view name and not the base table names when it describes columns from the view in the result set.

When a view is created by using WITH VIEW_METADATA, all its columns, except a **timestamp** column, are updatable if the view has INSERT or UPDATE INSTEAD OF triggers. For more information, see the Remarks section in CREATE VIEW.

AS

Are the actions the view is to take.

select_statement

Is the SELECT statement that defines the view.

WITH CHECK OPTION

Forces all data modification statements that are executed against the view to follow the criteria set within select_statement.

Remarks

For more information about ALTER VIEW, see Remarks in CREATE VIEW.

Note
If the previous view definition was created by using WITH ENCRYPTION or CHECK OPTION, these options are enabled only if they are included in ALTER VIEW.

If a view currently used is modified by using ALTER VIEW, the Database Engine takes an exclusive schema lock on the view. When the lock is granted, and there are no active users of the view, the Database Engine deletes all copies of the view from the procedure cache. Existing plans referencing the view remain in the cache but are recompiled when invoked.

ALTER VIEW can be applied to indexed views; however, ALTER VIEW unconditionally drops all indexes on the view.

Permissions

To execute ALTER VIEW, at a minimum, ALTER permission on OBJECT is required.

Examples

The following example creates a view that contains all employees and their hire dates called `EmployeeHireDate`. Permissions are granted to the view, but requirements are changed to select employees whose hire dates fall before a certain date. Then, `ALTER VIEW` is used to replace the view.

```
USE AdventureWorks2012 ;
GO
CREATE VIEW HumanResources.EmployeeHireDate
AS
```

```
SELECT p.FirstName, p.LastName, e.HireDate
FROM HumanResources.Employee AS e JOIN Person.Person AS  p
ON e.BusinessEntityID = p.BusinessEntityID ;
GO
```

The view must be changed to include only the employees that were hired before `2002`. If ALTER VIEW is not used, but instead the view is dropped and re-created, the previously used GRANT statement and any other statements that deal with permissions pertaining to this view must be re-entered.

```
ALTER VIEW HumanResources.EmployeeHireDate
AS
SELECT p.FirstName, p.LastName, e.HireDate
FROM HumanResources.Employee AS e JOIN Person.Person AS  p
ON e.BusinessEntityID = p.BusinessEntityID
WHERE HireDate < CONVERT(DATETIME,'20020101',101) ;
GO
```

See Also

CREATE TABLE
CREATE VIEW
DROP VIEW
Create a Stored Procedure
SELECT
EVENTDATA
Making Schema Changes on Publication Databases

ALTER WORKLOAD GROUP

Changes an existing Resource Governor workload group configuration, and optionally assigns it to a to a Resource Governor resource pool.

 Transact-SQL Syntax Conventions.

Syntax

```
ALTER WORKLOAD GROUP { group_name | "default" }
[ WITH
      ([ IMPORTANCE = { LOW | MEDIUM | HIGH } ]
        [ [ , ] REQUEST_MAX_MEMORY_GRANT_PERCENT = value ]
        [ [ , ] REQUEST_MAX_CPU_TIME_SEC = value ]
```

[[,] REQUEST_MEMORY_GRANT_TIMEOUT_SEC = **value**]
[[,] MAX_DOP = **value**]
[[,] GROUP_MAX_REQUESTS = **value**] **)**
]
[USING { **pool_name** | **"**default**"** }]
[;]

Arguments

group_name | "default"

Is the name of an existing user-defined workload group or the Resource Governor default workload group.

Note
Resource Governor creates the "default" and internal groups when SQL Server is installed.

The option "default" must be enclosed by quotation marks ("") or brackets ([]) when used with ALTER WORKLOAD GROUP to avoid conflict with DEFAULT, which is a system reserved word. For more information, see Database Identifiers.

Note
Predefined workload groups and resource pools all use lowercase names, such as "default". This should be taken into account for servers that use case-sensitive collation. Servers with case-insensitive collation, such as SQL_Latin1_General_CP1_CI_AS, will treat "default" and "Default" as the same.

IMPORTANCE = { LOW | MEDIUM | HIGH }

Specifies the relative importance of a request in the workload group. Importance is one of the following:

- LOW
- MEDIUM (default)
- HIGH

Note
Internally each importance setting is stored as a number that is used for calculations.

IMPORTANCE is local to the resource pool; workload groups of different importance inside the same resource pool affect each other, but do not affect workload groups in another resource pool.

REQUEST_MAX_MEMORY_GRANT_PERCENT = value

Specifies the maximum amount of memory that a single request can take from the pool. This percentage is relative to the resource pool size specified by MAX_MEMORY_PERCENT.

Note
The amount specified only refers to query execution grant memory.

value must be 0 or a positive integer. The allowed range for value is from 0 through 100. The default setting for value is 25.

Note the following:

- Setting value to 0 prevents queries with SORT and HASH JOIN operations in user-defined workload groups from running.
- We do not recommend setting value greater than 70 because the server may be unable to set aside enough free memory if other concurrent queries are running. This may eventually lead to query time-out error 8645.

Note

- If the query memory requirements exceed the limit that is specified by this parameter, the server does the following:
- For user-defined workload groups, the server tries to reduce the query degree of parallelism until the memory requirement falls under the limit, or until the degree of parallelism equals 1. If the query memory requirement is still greater than the limit, error 8657 occurs.
- For internal and default workload groups, the server permits the query to obtain the required memory.
- Be aware that both cases are subject to time-out error 8645 if the server has insufficient physical memory.

REQUEST_MAX_CPU_TIME_SEC = value

Specifies the maximum amount of CPU time, in seconds, that a request can use. value must be 0 or a positive integer. The default setting for value is 0, which means unlimited.

Note

Resource Governor will not prevent a request from continuing if the maximum time is exceeded. However, an event will be generated. For more information, see CPU Threshold Exceeded Event Class.

REQUEST_MEMORY_GRANT_TIMEOUT_SEC = value

Specifies the maximum time, in seconds, that a query can wait for memory grant (work buffer memory) to become available.

Note

A query does not always fail when memory grant time-out is reached. A query will only fail if there are too many concurrent queries running. Otherwise, the query may only get the minimum memory grant, resulting in reduced query performance.

value must be a positive integer. The default setting for value, 0, uses an internal calculation based on query cost to determine the maximum time.

MAX_DOP = value

Specifies the maximum degree of parallelism (DOP) for parallel requests. value must be 0 or a

positive integer, 1 though 255. When value is 0, the server chooses the max degree of parallelism. This is the default and recommended setting.

Note

The actual value that the Database Engine sets for MAX_DOP by might be less than the specified value. The final value is determined by the formula min(255, *number of CPUs)*.

Caution

Changing MAX_DOP can adversely affect a server's performance. If you must change MAX_DOP, we recommend that it be set to a value that is less than or equal to the maximum number of hardware schedulers that are present in a single NUMA node. We recommend that you do not set MAX_DOP to a value greater than 8.

MAX_DOP is handled as follows:

- MAX_DOP as a query hint is honored as long as it does not exceed workload group MAX_DOP.
- MAX_DOP as a query hint always overrides sp_configure 'max degree of parallelism'.
- Workload group MAX_DOP overrides sp_configure 'max degree of parallelism'.
- If the query is marked as serial (MAX_DOP = 1) at compile time, it cannot be changed back to parallel at run time regardless of the workload group or sp_configure setting.

After DOP is configured, it can only be lowered on grant memory pressure. Workload group reconfiguration is not visible while waiting in the grant memory queue.

GROUP_MAX_REQUESTS = value

Specifies the maximum number of simultaneous requests that are allowed to execute in the workload group. value must be 0 or a positive integer. The default setting for value, 0, allows unlimited requests. When the maximum concurrent requests are reached, a user in that group can log in, but is placed in a wait state until concurrent requests are dropped below the value specified.

USING { pool_name | "default" }

Associates the workload group with the user-defined resource pool identified by pool_name, which in effect puts the workload group in the resource pool. If pool_name is not provided or if the USING argument is not used, the workload group is put in the predefined Resource Governor default pool.

The option "default" must be enclosed by quotation marks ("") or brackets ([]) when used with ALTER WORKLOAD GROUP to avoid conflict with DEFAULT, which is a system reserved word. For more information, see Database Identifiers.

Note

The option "default" is case-sensitive.

Remarks

ALTER WORKLOAD GROUP is allowed on the default group.

Changes to the workload group configuration do not take effect until after ALTER RESOURCE GOVERNOR RECONFIGURE is executed.

When you are executing DDL statements, we recommend that you be familiar with Resource Governor states. For more information, see Resource Governor.

REQUEST_MEMORY_GRANT_PERCENT: In SQL Server 2005, index creation is allowed to use more workspace memory than initially granted for improved performance. This special handling is supported by Resource Governor in SQL Server 2012. However, the initial grant and any additional memory grant are limited by resource pool and workload group settings.

Index Creation on a Partitioned Table

The memory consumed by index creation on non-aligned partitioned table is proportional to the number of partitions involved. If the total required memory exceeds the per-query limit (REQUEST_MAX_MEMORY_GRANT_PERCENT) imposed by the Resource Governor workload group setting, this index creation may fail to execute. Because the "default" workload group allows a query to exceed the per-query limit with the minimum required memory to start for SQL Server 2005 compatibility, the user may be able to run the same index creation in "default" workload group, if the "default" resource pool has enough total memory configured to run such query.

Permissions

Requires CONTROL SERVER permission.

Examples

The following example shows how to change the importance of requests in the default group from `MEDIUM` to `LOW`.

```
ALTER WORKLOAD GROUP "default"
WITH (IMPORTANCE = LOW)
GO
ALTER RESOURCE GOVERNOR RECONFIGURE
GO
```

The following example shows how to move a workload group from the pool that it is in to the default pool.

```
ALTER WORKLOAD GROUP adHoc
USING [default];
GO
ALTER RESOURCE GOVERNOR RECONFIGURE
GO
```

See Also

Resource Governor

CREATE WORKLOAD GROUP (Transact-SQL)
DROP WORKLOAD GROUP (Transact-SQL)
CREATE RESOURCE POOL (Transact-SQL)
ALTER RESOURCE POOL (Transact-SQL)
DROP RESOURCE POOL (Transact-SQL)
ALTER RESOURCE GOVERNOR (Transact-SQL)

ALTER XML SCHEMA COLLECTION

Adds new schema components to an existing XML schema collection.

 Transact-SQL Syntax Conventions

Syntax

ALTER XML SCHEMA COLLECTION [**relational_schema.**]**sql_identifier** ADD '**Schema Component**'

Arguments

relational_schema

Identifies the relational schema name. If not specified, the default relational schema is assumed.

sql_identifier

Is the SQL identifier for the XML schema collection.

'Schema Component'

Is the schema component to insert.

Remarks

Use the ALTER XML SCHEMA COLLECTION to add new XML schemas whose namespaces are not already in the XML schema collection, or add new components to existing namespaces in the collection.

The following example adds a new <element> to the existing namespace `http://MySchema/test_xml_schema` in the collection `MyColl`.

```
-- First create an XML schema collection.
CREATE XML SCHEMA COLLECTION MyColl AS '
   <schema
    xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://MySchema/test_xml_schema">
      <element name="root" type="string"/>
```

```
  </schema>'
-- Modify the collection.
ALTER XML SCHEMA COLLECTION MyColl ADD '
  <schema xmlns="http://www.w3.org/2001/XMLSchema"
          targetNamespace="http://MySchema/test_xml_schema">
     <element name="anotherElement" type="byte"/>
 </schema>'
```

`ALTER XML SCHEMA` adds element `<anotherElement>` to the previously defined namespace `http://MySchema/test_xml_schema`.

Note that if some of the components you want to add in the collection reference components that are already in the same collection, you must use `<import namespace="referenced_component_namespace" />`. However, it is not valid to use the current schema namespace in `<xsd:import>`, and therefore components from the same target namespace as the current schema namespace are automatically imported.

To remove collections, use Guidelines and Limitations in Using XML Schema Collections on the Server.

If the schema collection already contains a lax validation wildcard or an element of type **xs:anyType**, adding a new global element, type, or attribute declaration to the schema collection will cause a revalidation of all the stored data that is constrained by the schema collection.

Permissions

To alter an XML SCHEMA COLLECTION requires ALTER permission on the collection.

Examples

A. Creating XML schema collection in the database

The following example creates the XML schema collection `ManuInstructionsSchemaCollection`. The collection has only one schema namespace.

```
-- Create a sample database in which to load the XML schema collection.
CREATE DATABASE SampleDB
GO
USE SampleDB
GO
CREATE XML SCHEMA COLLECTION ManuInstructionsSchemaCollection AS
N'<?xml version="1.0" encoding="UTF-16"?>
<xsd:schema
targetNamespace="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelManuInstructions"
```

```
   xmlns          ="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions"
   elementFormDefault="qualified"
   attributeFormDefault="unqualified"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema" >

    <xsd:complexType name="StepType" mixed="true" >
        <xsd:choice  minOccurs="0" maxOccurs="unbounded" >
            <xsd:element name="tool" type="xsd:string" />
            <xsd:element name="material" type="xsd:string" />
            <xsd:element name="blueprint" type="xsd:string" />
            <xsd:element name="specs" type="xsd:string" />
            <xsd:element name="diag" type="xsd:string" />
        </xsd:choice>
    </xsd:complexType>

    <xsd:element  name="root">
        <xsd:complexType mixed="true">
            <xsd:sequence>
                <xsd:element name="Location" minOccurs="1"
maxOccurs="unbounded">
                    <xsd:complexType mixed="true">
                        <xsd:sequence>
                            <xsd:element name="step" type="StepType"
minOccurs="1" maxOccurs="unbounded" />
                        </xsd:sequence>
                        <xsd:attribute name="LocationID" type="xsd:integer"
use="required"/>
                        <xsd:attribute name="SetupHours" type="xsd:decimal"
use="optional"/>
                        <xsd:attribute name="MachineHours" type="xsd:decimal"
use="optional"/>
                        <xsd:attribute name="LaborHours" type="xsd:decimal"
use="optional"/>
```

```
                        <xsd:attribute name="LotSize" type="xsd:decimal"
use="optional"/>
                    </xsd:complexType>
                </xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>' ;
GO
-- Verify - list of collections in the database.
SELECT *
FROM sys.xml_schema_collections
-- Verify - list of namespaces in the database.
SELECT name
FROM sys.xml_schema_namespaces

-- Use it. Create a typed xml variable. Note the collection name
-- that is specified.
DECLARE @x xml (ManuInstructionsSchemaCollection)
GO
--Or create a typed xml column.
CREATE TABLE T (
        i int primary key,
        x xml (ManuInstructionsSchemaCollection))
GO
-- Clean up.
DROP TABLE T
GO
DROP XML SCHEMA COLLECTION ManuInstructionsSchemaCollection
Go
USE master
GO
DROP DATABASE SampleDB
```

Alternatively, you can assign the schema collection to a variable and specify the variable in the `CREATE XML SCHEMA COLLECTION` statement as follows:

```
DECLARE @MySchemaCollection nvarchar(max)
Set @MySchemaCollection  = N' copy the schema collection here'
CREATE XML SCHEMA COLLECTION AS @MySchemaCollection
```

The variable in the example is of `nvarchar(max)` type. The variable can also be of **xml** data type, in which case, it is implicitly converted to a string.

For more information, see Viewing Stored XML Schema.

You can store schema collections in an **xml** type column. In this case, to create XML schema collection, perform the following steps:

1. Retrieve the schema collection from the column by using a SELECT statement and assign it to a variable of **xml** type, or a **varchar** type.
2. Specify the variable name in the CREATE XML SCHEMA COLLECTION statement.

The CREATE XML SCHEMA COLLECTION stores only the schema components that SQL Server understands; everything in the XML schema is not stored in the database. Therefore, if you want the XML schema collection back exactly the way it was supplied, we recommend that you save your XML schemas in a database column or some other folder on your computer.

B. Specifying multiple schema namespaces in a schema collection

You can specify multiple XML schemas when you create an XML schema collection. For example:

```
CREATE XML SCHEMA COLLECTION N'
<xsd:schema>....</xsd:schema>
<xsd:schema>...</xsd:schema>'
```

The following example creates the XML schema collection `ProductDescriptionSchemaCollection` that includes two XML schema namespaces.

```
CREATE XML SCHEMA COLLECTION ProductDescriptionSchemaCollection AS
'<xsd:schema
targetNamespace="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelWarrAndMain"
    xmlns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelWarrAndMain"
    elementFormDefault="qualified"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
    <xsd:element name="Warranty"  >
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="WarrantyPeriod" type="xsd:string"  />
```

```
                <xsd:element name="Description" type="xsd:string"  />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
 <xs:schema
targetNamespace="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription"
    xmlns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription"
    elementFormDefault="qualified"
    xmlns:mstns="http://tempuri.org/XMLSchema.xsd"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:wm="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain" >
    <xs:import
namespace="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain" />
    <xs:element name="ProductDescription" type="ProductDescription" />
        <xs:complexType name="ProductDescription">
            <xs:sequence>
                <xs:element name="Summary" type="Summary" minOccurs="0" />
            </xs:sequence>
            <xs:attribute name="ProductModelID" type="xs:string" />
            <xs:attribute name="ProductModelName" type="xs:string" />
        </xs:complexType>
        <xs:complexType name="Summary" mixed="true" >
            <xs:sequence>
                <xs:any processContents="skip"
namespace="http://www.w3.org/1999/xhtml" minOccurs="0" maxOccurs="unbounded"
/>
            </xs:sequence>
        </xs:complexType>
</xs:schema>'
;
```

```
GO
-- Clean up
DROP XML SCHEMA COLLECTION ProductDescriptionSchemaCollection
GO
```

C. Importing a schema that does not specify a target namespace

If a schema that does not contain a **targetNamespace** attribute is imported in a collection, its components are associated with the empty string target namespace as shown in the following example. Note that not associating one or more schemas imported in the collection results in multiple schema components (potentially unrelated) being associated with the default empty string namespace.

```
-- Create a collection that contains a schema with no target namespace.
CREATE XML SCHEMA COLLECTION MySampleCollection AS '
<schema xmlns="http://www.w3.org/2001/XMLSchema"  xmlns:ns="http://ns">
<element name="e" type="dateTime"/>
</schema>'
GO
-- query will return the names of all the collections that
--contain a schema with no target namespace
SELECT sys.xml_schema_collections.name
FROM   sys.xml_schema_collections
JOIN   sys.xml_schema_namespaces
ON     sys.xml_schema_collections.xml_collection_id =
       sys.xml_schema_namespaces.xml_collection_id
WHERE  sys.xml_schema_namespaces.name=''
```

See Also

CREATE XML SCHEMA COLLECTION
DROP XML SCHEMA COLLECTION (Transact-SQL)
EVENTDATA (Transact-SQL)
Typed vs. Untyped XML
Guidelines and Limitations of XML Schemas on the Server

CREATE Statements

SQL Server Transact-SQL contains the following CREATE statements. Use CREATE statements to define new entities. For example, use CREATE TABLE to add a new table to a database.

In this Section

CREATE AGGREGATE	CREATE FULLTEXT INDEX	CREATE SEARCH PROPERTY LIST (Transact-SQL)
CREATE APPLICATION ROLE	CREATE FULLTEXT STOPLIST	CREATE SEQUENCE (Transact-SQL)
CREATE ASSEMBLY	CREATE FUNCTION	CREATE SERVER AUDIT
CREATE ASYMMETRIC KEY	CREATE INDEX	CREATE SERVER AUDIT SPECIFICATION
CREATE BROKER PRIORITY	CREATE LOGIN	CREATE SERVICE
CREATE CERTIFICATE	CREATE MASTER KEY	CREATE SPATIAL INDEX
CREATE COLUMNSTORE INDEX	CREATE MESSAGE TYPE	CREATE STATISTICS
CREATE CONTRACT	CREATE PARTITION FUNCTION	CREATE SYMMETRIC KEY
CREATE CREDENTIAL	CREATE PARTITION SCHEME	CREATE SYNONYM
CREATE CRYPTOGRAPHIC PROVIDER	CREATE PROCEDURE	CREATE TABLE
CREATE DATABASE	CREATE QUEUE	CREATE TRIGGER
CREATE DATABASE AUDIT SPECIFICATION	CREATE REMOTE SERVICE BINDING	CREATE TYPE
CREATE DATABASE ENCRYPTION KEY	CREATE RESOURCE POOL	CREATE USER
CREATE DEFAULT	CREATE ROLE	CREATE VIEW
CREATE ENDPOINT	CREATE ROUTE	CREATE WORKLOAD GROUP
CREATE EVENT NOTIFICATION	CREATE RULE	CREATE XML INDEX
CREATE EVENT SESSION	CREATE SCHEMA	CREATE XML SCHEMA COLLECTION
CREATE FULLTEXT CATALOG		

See Also

ALTER Statements (Transact-SQL)

DROP Statements

CREATE AGGREGATE

Creates a user-defined aggregate function whose implementation is defined in a class of an assembly in the .NET Framework. For the Database Engine to bind the aggregate function to its implementation, the .NET Framework assembly that contains the implementation must first be uploaded into an instance of SQL Server by using a CREATE ASSEMBLY statement.

Note

By default, the ability of SQL Server to run CLR code is off. You can create, modify, and drop database objects that reference managed code modules, but the code in these modules will not run in an instance of SQL Server unless the clr enabled option is enabled by using sp_configure.

Transact-SQL Syntax Conventions

Syntax

```
CREATE AGGREGATE [ schema_name . ] aggregate_name
    (@param_name <input_sqltype>
    [ ,...n ] )
RETURNS <return_sqltype>
EXTERNAL NAME assembly_name [ .class_name ]

<input_sqltype> ::=
    system_scalar_type | { [ udt_schema_name. ] udt_type_name }

<return_sqltype> ::=
    system_scalar_type | { [ udt_schema_name. ] udt_type_name }
```

Arguments

schema_name

Is the name of the schema to which the user-defined aggregate function belongs.

aggregate_name

Is the name of the aggregate function you want to create.

@param_name

One or more parameters in the user-defined aggregate. The value of a parameter must be supplied by the user when the aggregate function is executed. Specify a parameter name by using an "at" sign (**@**) as the first character. The parameter name must comply with the rules for identifiers. Parameters are local to the function.

system_scalar_type

Is any one of the SQL Server system scalar data types to hold the value of the input parameter or return value. All scalar data types can be used as a parameter for a user-defined aggregate, except **text**, **ntext**, and **image**. Nonscalar types, such as **cursor** and **table**, cannot be specified.

udt_schema_name

Is the name of the schema to which the CLR user-defined type belongs. If not specified, the Database Engine references udt_type_name in the following order:

- The native SQL type namespace.
- The default schema of the current user in the current database.
- The **dbo** schema in the current database.

udt_type_name

Is the name of a CLR user-defined type already created in the current database. If udt_schema_name is not specified, SQL Server assumes the type belongs to the schema of the current user.

assembly_name [.class_name]

Specifies the assembly to bind with the user-defined aggregate function and, optionally, the name of the schema to which the assembly belongs and the name of the class in the assembly that implements the user-defined aggregate. The assembly must already have been created in the database by using a CREATE ASSEMBLY statement. class_name must be a valid SQL Server identifier and match the name of a class that exists in the assembly. class_name may be a namespace-qualified name if the programming language used to write the class uses namespaces, such as C#. If class_name is not specified, SQL Server assumes it is the same as aggregate_name.

Remarks

The class of the assembly referenced in assembly_name and its methods, should satisfy all the requirements for implementing a user-defined aggregate function in an instance of SQL Server. For more information, see DROP AGGREGATE (Transact-SQL).

Permissions

Requires CREATE AGGREGATE permission and also REFERENCES permission on the assembly that is specified in the EXTERNAL NAME clause.

Examples

The following example assumes that a StringUtilities.csproj sample application is compiled. For more information, see String Utilities Sample.

The example creates aggregate `Concatenate`. Before the aggregate is created, the assembly `StringUtilities.dll` is registered in the local database.

```
USE AdventureWorks;
GO
DECLARE @SamplesPath nvarchar(1024)
-- You may have to modify the value of the this variable if you have
--installed the sample some location other than the default location.
SELECT @SamplesPath = REPLACE(physical_name, 'Microsoft SQL
Server\MSSQL11.MSSQLSERVER\MSSQL\DATA\master.mdf', 'Microsoft SQL
Server\90\Samples\Engine\Programmability\CLR\')
     FROM master.sys.database_files
     WHERE name = 'master';
CREATE ASSEMBLY StringUtilities FROM @SamplesPath +
'StringUtilities\CS\StringUtilities\bin\debug\StringUtilities.dll'
WITH PERMISSION_SET=SAFE;
GO

CREATE AGGREGATE Concatenate(@input nvarchar(4000))
RETURNS nvarchar(4000)
EXTERNAL NAME [StringUtilities].[Microsoft.Samples.SqlServer.Concatenate];
GO
```

See Also

DROP AGGREGATE (Transact-SQL)

CREATE APPLICATION ROLE

Adds an application role to the current database.

Transact-SQL Syntax Conventions

Syntax

CREATE APPLICATION ROLE **application_role_name**
WITH PASSWORD = '**password**' [, DEFAULT_SCHEMA = **schema_name**]

Arguments

application_role_name

Specifies the name of the application role. This name must not already be used to refer to any principal in the database.

PASSWORD = 'password'

Specifies the password that database users will use to activate the application role. You should always use strong passwords. password must meet the Windows password policy requirements of the computer that is running the instance of SQL Server.

DEFAULT_SCHEMA = schema_name

Specifies the first schema that will be searched by the server when it resolves the names of objects for this role. If DEFAULT_SCHEMA is left undefined, the application role will use DBO as its default schema. schema_name can be a schema that does not exist in the database.

Remarks

Important

Password complexity is checked when application role passwords are set. Applications that invoke application roles must store their passwords. Application role passwords should always be stored encrypted.

Application roles are visible in the sys.database_principals catalog view.

For information about how to use application roles, see Application Roles.

Caution

Beginning with SQL Server 2005, the behavior of schemas changed. As a result, code that assumes that schemas are equivalent to database users may no longer return correct results. Old catalog views, including , should not be used in a database in which any of the following DDL statements have ever been used: CREATE SCHEMA, ALTER SCHEMA, DROP SCHEMA, CREATE USER, ALTER USER, DROP USER, CREATE ROLE, ALTER ROLE, DROP ROLE, CREATE APPROLE, ALTER APPROLE, DROP APPROLE, ALTER AUTHORIZATION. In such databases you must instead use the new catalog views. The new catalog views take into account the separation of principals and schemas that was introduced in SQL Server 2005. For more information about catalog views, see .

Permissions

Requires ALTER ANY APPLICATION ROLE permission on the database.

Examples

The following example creates an application role called `weekly_receipts` that has the password `987Gbv876sPYY5m23` and `Sales` as its default schema.

```
CREATE APPLICATION ROLE weekly_receipts
    WITH PASSWORD = '987G^bv876sPY)Y5m23'
    , DEFAULT_SCHEMA = Sales;
GO
```

See Also

Application Roles
sp_setapprole (Transact-SQL)
ALTER APPLICATION ROLE (Transact-SQL)
DROP APPLICATION ROLE (Transact-SQL)
Password Complexity and Expiration
eventdata (Transact-SQL)

CREATE ASSEMBLY

Creates a managed application module that contains class metadata and managed code as an object in an instance of SQL Server. By referencing this module, common language runtime (CLR) functions, stored procedures, triggers, user-defined aggregates, and user-defined types can be created in the database.

 Transact-SQL Syntax Conventions

Syntax

```
CREATE ASSEMBLY assembly_name
[ AUTHORIZATION owner_name ]
FROM { <client_assembly_specifier> | <assembly_bits> [ ,...n ] }
[ WITH PERMISSION_SET = { SAFE | EXTERNAL_ACCESS | UNSAFE } ]
[ ; ]
<client_assembly_specifier> ::=
      '[\\computer_name\]share_name\[path\]manifest_file_name'
  | '[local_path\]manifest_file_name'

<assembly_bits> ::=
{ varbinary_literal | varbinary_expression }
```

Arguments

assembly_name

Is the name of the assembly. The name must be unique within the database and a valid

identifier.

AUTHORIZATION owner_name

Specifies the name of a user or role as owner of the assembly. owner_name must either be the name of a role of which the current user is a member, or the current user must have IMPERSONATE permission on owner_name. If not specified, ownership is given to the current user.

<client_assembly_specifier>

Specifies the local path or network location where the assembly that is being uploaded is located, and also the manifest file name that corresponds to the assembly. <client_assembly_specifier> can be expressed as a fixed string or an expression evaluating to a fixed string, with variables. CREATE ASSEMBLY does not support loading multimodule assemblies. SQL Server also looks for any dependent assemblies of this assembly in the same location and also uploads them with the same owner as the root level assembly. If these dependent assemblies are not found and they are not already loaded in the current database, CREATE ASSEMBLY fails. If the dependent assemblies are already loaded in the current database, the owner of those assemblies must be the same as the owner of the newly created assembly.

<client_assembly_specifier> cannot be specified if the logged in user is being impersonated.

<assembly_bits>

Is the list of binary values that make up the assembly and its dependent assemblies. The first value in the list is considered the root-level assembly. The values corresponding to the dependent assemblies can be supplied in any order. Any values that do not correspond to dependencies of the root assembly are ignored.

Note

This option is not available in a contained database.

varbinary_literal

Is a **varbinary** literal.

varbinary_expression

Is an expression of type **varbinary**.

PERMISSION_SET { SAFE | EXTERNAL_ACCESS | UNSAFE }

Specifies a set of code access permissions that are granted to the assembly when it is accessed by SQL Server. If not specified, SAFE is applied as the default.

We recommend using SAFE. SAFE is the most restrictive permission set. Code executed by an assembly with SAFE permissions cannot access external system resources such as files, the network, environment variables, or the registry.

EXTERNAL_ACCESS enables assemblies to access certain external system resources such as files, networks, environmental variables, and the registry.

Note

This option is not available in a contained database.

UNSAFE enables assemblies unrestricted access to resources, both within and outside an instance of SQL Server. Code running from within an UNSAFE assembly can call unmanaged code.

Note

This option is not available in a contained database.

noteDXDOC112778PADS Security Note

- SAFE is the recommended permission setting for assemblies that perform computation and data management tasks without accessing resources outside an instance of SQL Server.
- We recommend using EXTERNAL_ACCESS for assemblies that access resources outside of an instance of SQL Server. EXTERNAL_ACCESS assemblies include the reliability and scalability protections of SAFE assemblies, but from a security perspective are similar to UNSAFE assemblies. This is because code in EXTERNAL_ACCESS assemblies runs by default under the SQL Server service account and accesses external resources under that account, unless the code explicitly impersonates the caller. Therefore, permission to create EXTERNAL_ACCESS assemblies should be granted only to logins that are trusted to run code under the SQL Server service account. For more information about impersonation, see EVENTDATA (Transact-SQL).
- Specifying UNSAFE enables the code in the assembly complete freedom to perform operations in the SQL Server process space that can potentially compromise the robustness of SQL Server. UNSAFE assemblies can also potentially subvert the security system of either SQL Server or the common language runtime. UNSAFE permissions should be granted only to highly trusted assemblies. Only members of the **sysadmin** fixed server role can create and alter UNSAFE assemblies.

For more information about assembly permission sets, see Designing Assemblies.

Remarks

CREATE ASSEMBLY uploads an assembly that was previously compiled as a .dll file from managed code for use inside an instance of SQL Server.

SQL Server does not allow registering different versions of an assembly with the same name, culture and public key.

When attempting to access the assembly specified in <client_assembly_specifier>, SQL Server impersonates the security context of the current Windows login. If <client_assembly_specifier> specifies a network location (UNC path), the impersonation of the current login is not carried forward to the network location because of delegation limitations. In this case, access is made using the security context of the SQL Server service account. For more information, see Credentials.

Besides the root assembly specified by assembly_name, SQL Server tries to upload any assemblies that are referenced by the root assembly being uploaded. If a referenced assembly is already uploaded to the database because of an earlier CREATE ASSEMBLY statement, this assembly is not uploaded but is available to the root assembly. If a dependent assembly was not previously uploaded, but SQL Server cannot locate its manifest file in the source directory, CREATE ASSEMBLY returns an error.

If any dependent assemblies referenced by the root assembly are not already in the database and are implicitly loaded together with the root assembly, they have the same permission set as the root level assembly. If the dependent assemblies must be created by using a different permission set than the root-level assembly, they must be uploaded explicitly before the root level assembly with the appropriate permission set.

Assembly Validation

SQL Server performs checks on the assembly binaries uploaded by the CREATE ASSEMBLY statement to guarantee the following:

- The assembly binary is well formed with valid metadata and code segments, and the code segments have valid Microsoft Intermediate language (MSIL) instructions.
- The set of system assemblies it references is one of the following supported assemblies in SQL Server: Microsoft.Visualbasic.dll, Mscorlib.dll, System.Data.dll, System.dll, System.Xml.dll, Microsoft.Visualc.dll, Custommarshallers.dll, System.Security.dll, System.Web.Services.dll, System.Data.SqlXml.dll, System.Core.dll, and System.Xml.Linq.dll. Other system assemblies can be referenced, but they must be explicitly registered in the database.
- For assemblies created by using SAFE or EXTERNAL ACCESS permission sets:
  - The assembly code should be type-safe. Type safety is established by running the common language runtime verifier against the assembly.
  - The assembly should not contain any static data members in its classes unless they are marked as read-only.
  - The classes in the assembly cannot contain finalizer methods.
  - The classes or methods of the assembly should be annotated only with allowed code attributes. For more information, see Custom Attributes for CLR Routines.

Besides the previous checks that are performed when CREATE ASSEMBLY executes, there are additional checks that are performed at execution time of the code in the assembly:

- Calling certain Microsoft .NET Framework APIs that require a specific Code Access Permission may fail if the permission set of the assembly does not include that permission.
- For SAFE and EXTERNAL_ACCESS assemblies, any attempt to call .NET Framework APIs that are annotated with certain HostProtectionAttributes will fail.

For more information, see Designing Assemblies.

Permissions

Requires CREATE ASSEMBLY permission.

If PERMISSION_SET = EXTERNAL_ACCESS is specified, the SQL Server login must have EXTERNAL ACCESS ASSEMBLY permission on the server. If PERMISSION_SET = UNSAFE is specified, membership in the **sysadmin** fixed server role is required.

User must be the owner of any assemblies that are referenced by the assembly that are to be uploaded if the assemblies already exist in the database. To upload an assembly by using a file path, the current user must be a Windows authenticated login or a member of the **sysadmin** fixed server role. The Windows login of the user that executes CREATE ASSEMBLY must have read permission on the share and the files being loaded in the statement.

For more information about assembly permission sets, see Designing Assemblies.

Examples

The following example assumes that the SQL Server Database Engine samples are installed in the default location of the local computer and the HelloWorld.csproj sample application is compiled. For more information, see Hello World Sample.

```
CREATE ASSEMBLY HelloWorld
FROM <system_drive>:\Program Files\Microsoft SQL
Server\100\Samples\HelloWorld\CS\HelloWorld\bin\debug\HelloWorld.dll
WITH PERMISSION_SET = SAFE;
```

See Also

ALTER ASSEMBLY
DROP ASSEMBLY
CREATE FUNCTION
CREATE PROCEDURE
CREATE TRIGGER
CREATE TYPE
CREATE AGGREGATE
EVENTDATA (Transact-SQL)
CLR Programmability Samples

CREATE ASYMMETRIC KEY

Creates an asymmetric key in the database.

 Transact-SQL Syntax Conventions

Syntax

CREATE ASYMMETRIC KEY **Asym_Key_Name**
  [AUTHORIZATION **database_principal_name**]

```
{
  [ FROM <Asym_Key_Source> ]
  |
  WITH <key_option>
[ ENCRYPTION BY <encrypting_mechanism> ]

<Asym_Key_Source>::=
  FILE = 'path_to_strong-name_file'
  |
  EXECUTABLE FILE = 'path_to_executable_file'
  |
  ASSEMBLY Assembly_Name
  |
  PROVIDER Provider_Name

<key_option> ::=
  ALGORITHM = <algorithm>
    |
  PROVIDER_KEY_NAME = 'key_name_in_provider'
    |
    CREATION_DISPOSITION = { CREATE_NEW | OPEN_EXISTING }

<algorithm> ::=
    { RSA_512 | RSA_1024 | RSA_2048 }

<encrypting_mechanism> ::=
   PASSWORD = 'password'
```

Arguments

FROM Asym_Key_Source

Specifies the source from which to load the asymmetric key pair.

AUTHORIZATION database_principal_name

Specifies the owner of the asymmetric key. The owner cannot be a role or a group. If this option is omitted, the owner will be the current user.

FILE = 'path_to_strong-name_file'

Specifies the path of a strong-name file from which to load the key pair.

Note

This option is not available in a contained database.

EXECUTABLE FILE = 'path_to_executable_file'

Specifies an assembly file from which to load the public key. Limited to 260 characters by MAX_PATH from the Windows API.

Note

This option is not available in a contained database.

ASSEMBLY Assembly_Name

Specifies the name of an assembly from which to load the public key.

ENCRYPTION BY <key_name_in_provider>

Specifies how the key is encrypted. Can be a certificate, password, or asymmetric key.

KEY_NAME = 'key_name_in_provider'

Specifies the key name from the external provider. For more information about external key management, see Understanding Extensible Key Management (EKM).

CREATION_DISPOSITION = CREATE_NEW

Creates a new key on the Extensible Key Management device. PROV_KEY_NAME must be used to specify key name on the device. If a key already exists on the device the statement fails with error.

CREATION_DISPOSITION = OPEN_EXISTING

Maps a SQL Server asymmetric key to an existing Extensible Key Management key. PROV_KEY_NAME must be used to specify key name on the device. If CREATION_DISPOSITION = OPEN_EXISTING is not provided, the default is CREATE_NEW.

PASSWORD = 'password'

Specifies the password with which to encrypt the private key. If this clause is not present, the private key will be encrypted with the database master key. password is a maximum of 128 characters. password must meet the Windows password policy requirements of the computer that is running the instance of SQL Server.

Remarks

An *asymmetric key* is a securable entity at the database level. In its default form, this entity contains both a public key and a private key. When executed without the FROM clause, CREATE ASYMMETRIC KEY generates a new key pair. When executed with the FROM clause, CREATE ASYMMETRIC KEY imports a key pair from a file or imports a public key from an assembly.

By default, the private key is protected by the database master key. If no database master key has been created, a password is required to protect the private key. If a database master key does exist, the password is optional.

The private key can be 512, 1024, or 2048 bits long.

Permissions

Requires CREATE ASYMMETRIC KEY permission on the database. If the AUTHORIZATION clause is specified, requires IMPERSONATE permission on the database principal, or ALTER permission on the application role. Only Windows logins, SQL Server logins, and application roles can own asymmetric keys. Groups and roles cannot own asymmetric keys.

Examples

A. Creating an asymmetric key

The following example creates an asymmetric key named `PacificSales09` by using the `RSA_2048` algorithm, and protects the private key with a password.

```
CREATE ASYMMETRIC KEY PacificSales09
    WITH ALGORITHM = RSA_2048
    ENCRYPTION BY PASSWORD = '<enterStrongPasswordHere>';
GO
```

B. Creating an asymmetric key from a file, giving authorization to a user

The following example creates the asymmetric key `PacificSales19` from a key pair stored in a file, and then authorizes user `Christina` to use the asymmetric key.

```
CREATE ASYMMETRIC KEY PacificSales19 AUTHORIZATION Christina
    FROM FILE = 'c:\PacSales\Managers\ChristinaCerts.tmp'
    ENCRYPTION BY PASSWORD = '<enterStrongPasswordHere>';
GO
```

C. Creating an asymmetric key from an EKM provider

The following example creates the asymmetric key `EKM_askey1` from a key pair stored in a file. It then encrypts it using an Extensible Key Management provider called `EKMProvider1`, and a key on that provider called `key10_user1`.

```
CREATE ASYMMETRIC KEY EKM_askey1
    FROM PROVIDER EKM_Provider1
    WITH
        ALGORITHM = RSA_512,
        CREATION_DISPOSITION = CREATE_NEW
        , PROVIDER_KEY_NAME  = 'key10_user1' ;
GO
```

See Also

Encryption Hierarchy

ALTER ASYMMETRIC KEY (Transact-SQL)

DROP ASYMMETRIC KEY (Transact-SQL)
Encryption Hierarchy

CREATE AVAILABILITY GROUP

Creates a new availability group, if the instance of SQL Server is enabled for the AlwaysOn Availability Groups feature.

Important
Execute CREATE AVAILABILITY GROUP on the instance of SQL Server that you intend to use as the initial primary replica of your new availability group. This server instance must reside on a Windows Server Failover Clustering (WSFC) node.

Transact-SQL Syntax Conventions

Syntax

```
CREATE AVAILABILITY GROUP group_name
   WITH (<with_option_spec> [ ,...n ] )
   FOR [ DATABASE database_name [ ,...n ] ]
   REPLICA ON <add_replica_spec> [ ,...n ]
   [ LISTENER 'dns_name' ( <listener_option> ) ]
[ ; ]

<with_option_spec>::=
    AUTOMATED_BACKUP_PREFERENCE = { PRIMARY | SECONDARY_ONLY| SECONDARY | NONE
}
  | FAILURE_CONDITION_LEVEL  = { 1 | 2 | 3 | 4 | 5 }
  | HEALTH_CHECK_TIMEOUT = milliseconds

<add_replica_spec>::=
  <server_instance> WITH
    (
      ENDPOINT_URL = 'TCP://system-address:port',
      AVAILABILITY_MODE = { SYNCHRONOUS_COMMIT | ASYNCHRONOUS_COMMIT },
      FAILOVER_MODE = { AUTOMATIC | MANUAL }
      [ , <add_replica_option> [ ,...n ] ]
    )
```

```
<add_replica_option>::=
    BACKUP_PRIORITY = n
  | SECONDARY_ROLE ( {
        [ ALLOW_CONNECTIONS = { NO | READ_ONLY | ALL } ]
      [,] [ READ_ONLY_ROUTING_URL = 'TCP://system-address:port' ]
    } )
    | PRIMARY_ROLE ( {
        [ ALLOW_CONNECTIONS = { READ_WRITE | ALL } ]
      [,] [ READ_ONLY_ROUTING_LIST = { ( '<server_instance>' [ ,...n ] ) | NONE } ]
    } )
    | SESSION_TIMEOUT = integer

<listener_option> ::=
  {
    WITH DHCP [ ON ( <network_subnet_option> ) ]
   | WITH IP ( { ( <ip_address_option> ) } [ , ...n ] ) [ , PORT = listener_port ]
  }

  <network_subnet_option> ::=
    'four_part_ipv4_address', 'four_part_ipv4_mask'

  <ip_address_option> ::=
    {
      'four_part_ipv4_address', 'four_part_ipv4_mask'
     | 'ipv6_address'
    }
```

Arguments

group_name

Specifies the name of the new availability group. group_name must be a valid SQL Server identifier, and it must be unique across all availability groups in the WSFC cluster. The maximum length for an availability group name is 128 characters.

AUTOMATED_BACKUP_PREFERENCE = { PRIMARY | SECONDARY_ONLY| SECONDARY | NONE }

Specifies a preference about how a backup job should evaluate the primary replica when choosing where to perform backups. You can script a given backup job to take the

automated backup preference into account. It is important to understand that the preference is not enforced by SQL Server, so it has no impact on ad-hoc backups.

The supported values are as follows:

PRIMARY

Specifies that the backups should always occur on the primary replica. This option is useful if you need backup features, such as creating differential backups, that are not supported when backup is run on a secondary replica.

SECONDARY_ONLY

Specifies that backups should never be performed on the primary replica. If the primary replica is the only replica online, the backup should not occur.

SECONDARY

Specifies that backups should occur on a secondary replica except when the primary replica is the only replica online. In that case, the backup should occur on the primary replica. This is the default behavior.

NONE

Specifies that you prefer that backup jobs ignore the role of the availability replicas when choosing the replica to perform backups. Note backup jobs might evaluate other factors such as backup priority of each availability replica in combination with its operational state and connected state.

There is no enforcement of the AUTOMATED_BACKUP_PREFERENCE setting. The interpretation of this preference depends on the logic, if any, that you script into back jobs for the databases in a given availability group. For more information, see Backup on Secondary Replicas (AlwaysOn Availability Groups).

Note

To view the automated backup preference of an existing availability group, select the **automated_backup_preference** or **automated_backup_preference_desc** column of the sys.availability_groups catalog view.

FAILURE_CONDITION_LEVEL = { 1 | 2 | 3 | 4 | 5 }

Specifies what failure conditions will trigger an automatic failover for this availability group. FAILURE_CONDITION_LEVEL is set at the group level but is relevant only on availability replicas that are configured for synchronous-commit availability mode (AVAILIBILITY_MODE = SYNCHRONOUS_COMMIT). Furthermore, failure conditions can trigger an automatic failover only if both the primary and secondary replicas are configured for automatic failover mode (FAILOVER_MODE = AUTOMATIC) and the secondary replica is currently synchronized with the primary replica.

The failure-condition levels (1–5) range from the least restrictive, level 1, to the most restrictive, level 5. A given condition level encompasses all the less restrictive levels. Thus, the strictest condition level, 5, includes the four less restrictive condition levels (1-4), level 4

includes levels 1-3, and so forth. The following table describes the failure-condition that corresponds to each level.

Level	Failure Condition
1	Specifies that an automatic failover should be initiated when any of the following occurs: • The SQL Server service is down. • The lease of the availability group for connecting to the WSFC cluster expires because no ACK is received from the server instance.
2	Specifies that an automatic failover should be initiated when any of the following occurs: • The instance of SQL Server does not connect to cluster, and the user-specified HEALTH_CHECK_TIMEOUT threshold of the availability group is exceeded. • The availability replica is in failed state.
3	Specifies that an automatic failover should be initiated on critical SQL Server internal errors, such as orphaned spinlocks, serious write-access violations, or too much dumping. This is the default behavior.
4	Specifies that an automatic failover should be initiated on moderate SQL Server internal errors, such as a persistent out-of-memory condition in the SQL Server internal resource pool.
5	Specifies that an automatic failover should be initiated on any qualified failure conditions, including: • Exhaustion of SQL Engine worker-threads. • Detection of an unsolvable deadlock.

nNote

Lack of response by an instance of SQL Server to client requests is not relevant to availability groups.

The FAILURE_CONDITION_LEVEL and HEALTH_CHECK_TIMEOUT values, define a *flexible failover policy* for a given group. This flexible failover policy provides you with granular control over what conditions must cause an automatic failover. For more information, see Flexible Failover Policy for Automatic Failover of an Availability Group (SQL Server).

HEALTH_CHECK_TIMEOUT = milliseconds

Specifies the wait time (in milliseconds) for the sp_server_diagnostics system stored procedure to return server-health information before the WSFC cluster assumes that the server instance is slow or hung. HEALTH_CHECK_TIMEOUT is set at the group level but is relevant only on availability replicas that are configured for synchronous-commit availability mode with automatic failover (AVAILIBILITY_MODE = SYNCHRONOUS_COMMIT). Furthermore, a health-check timeout can trigger an automatic failover only if both the primary and secondary replicas are configured for automatic failover mode (FAILOVER_MODE = AUTOMATIC) and the secondary replica is currently synchronized with the primary replica.

The default HEALTH_CHECK_TIMEOUT value is 30000 milliseconds (30 seconds). The minimum value is 15000 milliseconds (15 seconds), and the maximum value is 4294967295 milliseconds.

Important

sp_server_diagnostics does not perform health checks at the database level.

DATABASE database_name

Specifies a list of one or more user databases on the local SQL Server instance (that is, the server instance on which you are creating the availability group). You can specify multiple databases for an availability group, but each database can belong to only one availability group. For information about the type of databases that an availability group can support, see Prerequisites, Restrictions, and Recommendations for AlwaysOn Availability Groups (SQL Server). To find out which local databases already belong to an availability group, see the **replica_id** column in the sys.databases catalog view.

The DATABASE clause is optional. If you omit it, the new availability group will be empty.

After you have created the availability group, you will need connect to each server instance that hosts a secondary replica and then prepare each secondary database and join it to the availability group. For more information, see Start Data Movement on an AlwaysOn Secondary Database (SQL Server).

Note

Later, you can add eligible databases on the server instance that hosts the current primary replica to an availability group. You can also remove a database from an availability group. For more

information, see ALTER AVAILABILITY GROUP (Transact-SQL).

REPLICA ON

Specifies from one to five SQL server instances to host availability replicas in the new availability group. Each replica is specified by its server instance address followed by a WITH (...) clause. Minimally, you must specify your local server instance, which will become the initial primary replica. Optionally, you can also specify up to four secondary replicas.

You need to join every secondary replica to the availability group. For more information, see ALTER AVAILABILITY GROUP (Transact-SQL).

Note

If you specify less than four secondary replicas when you create an availability group, you can an additional secondary replica at any time by using the ALTER AVAILABILITY GROUP Transact-SQL statement. You can also use this statement this remove any secondary replica from an existing availability group.

<server_instance>

Specifies the address of the instance of SQL Server that is the host for an replica. The address format depends on whether the instance is the default instance or a named instance and whether it is a standalone instance or a failover cluster instance (FCI), as follows:

{ 'system_name[\instance_name]' | 'FCI_network_name[\instance_name]' }

The components of this address are as follows:

system_name

Is the NetBIOS name of the computer system on which the target instance of SQL Server resides. This computer must be a WSFC node.

FCI_network_name

Is the network name that is used to access a SQL Server failover cluster. Use this if the server instance participates as a SQL Server failover partner. Executing SELECT @@SERVERNAME on an FCI server instance returns its entire 'FCI_network_name[\instance_name]' string (which is the full replica name).

instance_name

Is the name of an instance of a SQL Server that is hosted by system_name or FCI_network_name and that has HADR service is enabled. For a default server instance, instance_name is optional. The instance name is case insensitive. On a stand-alone server instance, this value name is the same as the value returned by executing SELECT @@SERVERNAME.

****

Is a separator used only when specifying instance_name, in order to separate it from system_name or FCI_network_name.

For information about the prerequisites for WSFC nodes and server instances, see

Prerequisites, Restrictions, and Recommendations for AlwaysOn Availability Groups (SQL Server).

ENDPOINT_URL = 'TCP://system-address:port'

Specifies the URL path for the database mirroring endpoint on the instance of SQL Server that will host the availability replica that you are defining in your current REPICA ON clause.

The ENDPOINT_URL clause is required. For more information, see Specify the Endpoint URL When Adding or Modifying an Availability Replica.

'TCP://system-address:port'

Specifies a URL for specifying an endpoint URL or read-only routing URL. The URL parameters are as follows:

system-address

Is a string, such as a system name, a fully qualified domain name, or an IP address, that unambiguously identifies the destination computer system.

port

Is a port number that is associated with the mirroring endpoint of the partner server instance (for the ENDPOINT_URL option) or the port number used by the Database Engine of the server instance (for the READ_ONLY_ROUTING_URL option).

AVAILABILITY_MODE = { SYNCHRONOUS_COMMIT | ASYNCHRONOUS_COMMIT }

Specifies whether the primary replica has to wait for the secondary replica to acknowledge the hardening (writing) of the log records to disk before the primary replica can commit the transaction on a given primary database. The transactions on different databases on the same primary replica can commit independently.

SYNCHRONOUS_COMMIT

Specifies that the primary replica will wait to commit transactions until they have been hardened on this secondary replica (synchronous-commit mode). You can specify SYNCHRONOUS_COMMIT for up to three replicas, including the primary replica.

ASYNCHRONOUS_COMMIT

Specifies that the primary replica commits transactions without waiting for this secondary replica to harden the log (synchronous-commit availability mode). You can specify ASYNCHRONOUS_COMMIT for up to five availability replicas, including the primary replica.

The AVAILABILITY_MODE clause is required. For more information, see Availability Modes (AlwaysOn Availability Groups).

FAILOVER_MODE = { AUTOMATIC | MANUAL }

Specifies the failover mode of the availability replica that you are defining.

AUTOMATIC

Enables automatic failover. This option is supported only if you also specify

AVAILABILITY_MODE = SYNCHRONOUS_COMMIT. You can specify AUTOMATIC for two availability replicas, including the primary replica.

Note

SQL Server Failover Cluster Instances (FCIs) do not support automatic failover by availability groups, so any availability replica that is hosted by an FCI can only be configured for manual failover.

MANUAL

Enables manual failover or forced manual failover (*forced failover*) by the database administrator.

The FAILOVER_MODE clause is required. Two types of manual failover exist, manual failover without data loss and forced failover (with possible data loss), which are supported under different conditions. For more information, see Failover Modes (AlwaysOn Availability Groups).

BACKUP_PRIORITY = n

Specifies your priority for performing backups on this replica relative to the other replicas in the same availability group. The value is an integer in the range of 0..100. These values have the following meanings:

- 1..100 indicates that the availability replica could be chosen for performing backups. 1 indicates the lowest priority, and 100 indicates the highest priority. If BACKUP_PRIORITY = 1, the availability replica would be chosen for performing backups only if no higher priority availability replicas are currently available.
- 0 indicates that this availability replica will never be chosen for performing backups. This is useful, for example, for a remote availability replica to which you never want backups to fail over.

For more information, see Backup on Secondary Replicas (AlwaysOn Availability Groups).

SECONDARY_ROLE (...)

Specifies role-specific settings that will take effect if this availability replica currently owns the secondary role (that is, whenever it is a secondary replica). Within the parentheses, specify either or both secondary-role options. If you specify both, use a comma-separated list.

The secondary role options are as follows:

ALLOW_CONNECTIONS = { NO | READ_ONLY | ALL }

Specifies whether the databases of a given availability replica that is performing the secondary role (that is, is acting as a secondary replica) can accept connections from clients, one of:

NO

No user connections are allowed to secondary databases of this replica. They are not available for read access. This is the default behavior.

READ_ONLY

Only connections are allowed to the databases in the secondary replica where the Application Intent property is set to **ReadOnly**. For more information about this property, see Using Connection String Keywords with SQL Server Native Client.

ALL

All connections are allowed to the databases in the secondary replica for read-only access.

For more information, see Read-Only Access to Secondary Replicas.

READ_ONLY_ROUTING_URL = 'TCP://system-address:port'

Specifies the URL to be used for routing read-intent connection requests to this availability replica. This is the URL on which the SQL Server Database Engine listens. Typically, the default instance of the SQL Server Database Engine listens on TCP port 1433.

For a named instance, you can obtain the port number by querying the **port** and **type_desc** columns of the sys.dm_tcp_listener_states dynamic management view. The server instance uses the Transact-SQL listener (**type_desc = 'TSQL'**).

Note

For a named instance of SQL Server, the Transact-SQL listener should be configured to use a specific port. For more information, see Configure a Server to Listen on a Specific TCP Port (SQL Server Configuration Manager).

PRIMARY_ROLE (...)

Specifies role-specific settings that will take effect if this availability replica currently owns the primary role (that is, whenever it is the primary replica). Within the parentheses, specify either or both primary-role options. If you specify both, use a comma-separated list.

The primary role options are as follows:

ALLOW_CONNECTIONS = { READ_WRITE | ALL }

Specifies the type of connection that the databases of a given availability replica that is performing the primary role (that is, is acting as a primary replica) can accept from clients, one of:

READ_WRITE

Connections where the Application Intent connection property is set to **ReadOnly** are disallowed. When the Application Intent property is set to **ReadWrite** or the Application Intent connection property is not set, the connection is allowed. For more information about Application Intent connection property, see Using Connection String Keywords with SQL Server Native Client.

ALL

All connections are allowed to the databases in the primary replica. This is the default behavior.

READ_ONLY_ROUTING_LIST = { ('<server_instance>' [,...n]) | NONE }

Specifies a comma-separated list of server instances that host availability replicas for this availability group that meet the following requirements when running under the secondary role:

- Be configured to allow all connections or read-only connections (see the ALLOW_CONNECTIONS argument of the SECONDARY_ROLE option, above).
- Have their read-only routing URL defined (see the READ_ONLY_ROUTING_URL argument of the SECONDARY_ROLE option, above).

The READ_ONLY_ROUTING_LIST values are as follows:

<server_instance>

Specifies the address of the instance of SQL Server that is the host for an availability replica that is a readable secondary replica when running under the secondary role.

Use a comma-separated list to specify all the server instances that might host a readable secondary replica. Read-only routing will follow the order in which server instances are specified in the list. If you include a replica's host server instance on the replica's read-only routing list, placing this server instance at the end of the list is typically a good practice, so that read-intent connections go to a secondary replica, if one is available.

NONE

Specifies that when this availability replica is the primary replica, read-only routing will not be supported. This is the default behavior.

SESSION_TIMEOUT = integer

Specifies the session-timeout period in seconds. If you do not specify this option, by default, the time period is 10 seconds. The minimum value is 5 seconds.

Important

We recommend that you keep the time-out period at 10 seconds or greater.

For more information about the session-timeout period, see Overview of AlwaysOn Availability Groups (SQL Server).

LISTENER 'dns_name' (<listener_option>)

Defines a new availability group listener for this availability group. LISTENER is an optional argument.

Important

- Before you create your first listener, we strongly recommend that you read Prerequisites, Restrictions, and Recommendations for AlwaysOn Client Connectivity (SQL Server).
- After you create a listener for a given availability group, we strongly recommend that you do the following:

dns_name

Specifies the DNS host name of the availability group listener. The DNS name of the listener must be unique in the domain and in NetBIOS.

dns_name is a string value. This name can contain only alphanumeric characters, dashes (-), and hyphens (_), in any order. DNS host names are case insensitive. The maximum length is 63 characters.

We recommend that you specify a meaningful string. For example, for an availability group named `AG1`, a meaningful DNS host name would be `ag1-listener`.

Important

NetBIOS recognizes only the first 15 chars in the dns_name. If you have two WSFC clusters that are controlled by the same Active Directory and you try to create availability group listeners in both of clusters using names with more than 15 characters and an identical 15 character prefix, you will get an error reporting that the Virtual Network Name resource could not be brought online. For information about prefix naming rules for DNS names, see Assigning Domain Names.

<listener_option>

LISTENER takes one of the following <listener_option> options:

WITH DHCP [ON { ('four_part_ipv4_address', 'four_part_ipv4_mask') }]

Specifies that the availability group listener will use the Dynamic Host Configuration Protocol (DHCP). Optionally, use the ON clause to identify the network on which this listener will be created. DHCP is limited to a single subnet that is used for every server instances that hosts an availability replica in the availability group.

Important

We do not recommend DHCP in production environment. If there is a down time and the DHCP IP lease expires, extra time is required to register the new DHCP network IP address that is associated with the listener DNS name and impact the client connectivity. However, DHCP is good for setting up your development and testing environment to verify basic functions of availability groups and for integration with your applications.

For example:

```
WITH DHCP ON ('10.120.19.0','255.255.254.0')
```

WITH IP ({ ('four_part_ipv4_address', 'four_part_ipv4_mask') | ('ipv6_address') } [, ...n]) [, PORT = listener_port]

Specifies that, instead of using DHCP, the availability group listener will use one or more static IP addresses. To create an availability group across multiple subnets, each subnet requires one static IP address in the listener configuration. For a given subnet, the static IP address can be either an IPv4 address or an IPv6 address. Contact your network administrator to get a static IP address for each subnet that will host an availability replica for the new availability group.

For example:

```
WITH IP ( ('10.120.19.155','255.255.254.0') )
```

four_part_ipv4_address

Specifies an IPv4 four-part address for an availability group listener. For example, `10.120.19.155`.

four_part_ipv4_mask

Specifies an IPv4 four-part mask for an availability group listener. For example, `255.255.254.0`.

ipv6_address

Specifies an IPv6 address for an availability group listener. For example, `2001::4898:23:1002:20f:1fff:feff:b3a3`.

PORT = listener_port

Specifies the port number—listener_port—to be used by an availability group listener that is specified by a WITH IP clause. PORT is optional.

The default port number, 1433, is supported. However, if you have security concerns, we recommend using a different port number.

For example: `WITH IP ( ('2001::4898:23:1002:20f:1fff:feff:b3a3') ) , PORT = 7777`

Prerequisites and Restrictions

For information about the prerequisites for creating an availability group, see Prerequisites, Restrictions, and Recommendatoions for AlwaysOn Availability Groups.

For information about restrictions on the AVAILABILITY GROUP Transact-SQL statements, see Overview of Transact-SQL Statements for AlwaysOn Availability Groups .

Security

Permissions

Requires membership in the **sysadmin** fixed server role and either CREATE AVAILABILITY GROUP server permission, ALTER ANY AVAILABILITY GROUP permission, or CONTROL SERVER permission.

Examples

A. Configuring Backup on Secondary Replicas, Flexible Failover Policy, and Connection Access

The following example creates an availability group named `MyAg` for two user databases, `ThisDatabase` and `ThatDatabase`. The following table summarizes the values specified for the options that are set for the availability group as a whole.

Group Option	Setting	Description
AUTOMATED_BACKUP_PREFERENCE	SECONDARY	This automated backup preference indicates that backups should occur on a secondary replica except when the primary replica is the only replica online (this is the default behavior). For the AUTOMATED_BACKUP_PREFERENCE setting to have any effect, you need to script backup jobs on the availability databases to take the automated backup preference into account.
FAILURE_CONDITION_LEVEL	3	This failure condition level setting specifies that an automatic failover should be initiated on critical SQL Server internal errors, such as orphaned spinlocks, serious write-access violations, or too much dumping.
HEALTH_CHECK_TIMEOUT	600000	This health check timeout value, 60 seconds, specifies that the WSFC cluster will wait 60000 milliseconds for the sp_server_diagnostics system stored procedure to return server-health information about a server instance that is hosting a synchronous-commit replica with automatic before the cluster assumes that the host server instance is slow or hung. (The default value is 30000 milliseconds).

Three availability replicas are to be hosted by the default server instances on computers named `COMPUTER01`, `COMPUTER02`, and `COMPUTER03`. The following table summarizes the values specified for the replica options of each replica.

Replica Option	Setting on COMPUTER01	Setting on COMPUTER02	Setting on COMPUTER03	Description
ENDPOINT_URL	TCP://*COMPUTER01:5022*	TCP://*COMPUTER02:5022*	TCP://*COMPUTER03:5022*	In this example, the systems are the same domain, so the endpoint URLs can use the name of the computer system as the system address.
AVAILABILITY_MODE	SYNCHRONOUS_COMMIT	SYNCHRONOUS_COMMIT	ASYNCHRONOUS_COMMIT	Two of the replicas use synchronous-commit mode. When synchronized, they support failover without data loss. The third replica, which uses asynchronous-commit availability mode.
FAILOVER_MO	AUTOMATIC	AUTOMATIC	MANUAL	The

Replica Option	Setting on COMPUTER01	Setting on COMPUTER02	Setting on COMPUTER03	Description
DE				synchronous-commit replicas support automatic failover and planned manual failover. The synchronous-commit availability mode replica supports only forced manual failover.
BACKUP_PRIORITY	30	30	90	A higher priority, 90, is assigned to the asynchronous-commit replica, than to the synchronous-commit replicas. Backups will tend to occur on the server

Replica Option	Setting on COMPUTER01	Setting on COMPUTER02	Setting on COMPUTER03	Description
				instance that hosts the asynchronous-commit replica.
SECONDARY_ROLE	(ALLOW_CONNECTIONS = NO, READ_ONLY_ROUTING_URL = 'TCP://COMPUTER01:1433')	(ALLOW_CONNECTIONS = NO, READ_ONLY_ROUTING_URL = 'TCP://COMPUTER02:1433')	(ALLOW_CONNECTIONS = READ_ONLY, READ_ONLY_ROUTING_URL = 'TCP://COMPUTER03:1433')	Only the asynchronous-commit replica serves as a readable secondary replica. Specifies the computer name and default Database Engine port number (1433). This argument is optional.
PRIMARY_ROLE	(ALLOW_CONNECTIONS = READ_WRITE, READ_ONLY_ROUTING_LIST = (COMPUTER03))	(ALLOW_CONNECTIONS = READ_WRITE, READ_ONLY_ROUTING_LIST = (COMPUTER03))	(ALLOW_CONNECTIONS = READ_WRITE, READ_ONLY_ROUTING_LIST = NONE)	In the primary role, all the replicas will reject read-intent connection

Replica Option	Setting on COMPUTER01	Setting on COMPUTER02	Setting on COMPUTER03	Description
				attempts. Read-intent connection requests will be routed to COMPUTER03 if the local replica is running under the secondary role. When that replica runs under the primary role, read-only routing is disabled. This argument is optional.
SESSION_TIMEOUT	10	10	10	This example specifies the default session timeout value (10). This argument is optional.

Finally, the example specifies the optional LISTENER clause to create an availability group listener for the new availability group. A unique DNS name, `MyAgListenerIvP6`, is specified for this listener. The two replicas are on different subnets, so the listener must use static IP addresses. For each of the two availability replicas, the WITH IP clause specifies a static IP address, `2001:4898:f0:f00f::cf3c` and `2001:4898:e0:f213::4ce2`, which use the IPv6 format. This example also specifies uses the optional PORT argument to specify port `60173` as the listener port.

```
CREATE AVAILABILITY GROUP MyAg
   WITH (
      AUTOMATED_BACKUP_PREFERENCE = SECONDARY,
      FAILURE_CONDITION_LEVEL  =  3,
      HEALTH_CHECK_TIMEOUT = 600000
       )

   FOR
      DATABASE  ThisDatabase, ThatDatabase
   REPLICA ON
      'COMPUTER01' WITH
         (
         ENDPOINT_URL = 'TCP://COMPUTER01:5022',
         AVAILABILITY_MODE = SYNCHRONOUS_COMMIT,
         FAILOVER_MODE = AUTOMATIC,
         BACKUP_PRIORITY = 30,
         SECONDARY_ROLE (ALLOW_CONNECTIONS = NO,
            READ_ONLY_ROUTING_LIST = (COMPUTER03) ),
         PRIMARY_ROLE (ALLOW_CONNECTIONS = READ_WRITE,
            READ_ONLY_ROUTING_LIST = (COMPUTER03) ),
         SESSION_TIMEOUT = 10
         ),

      'COMPUTER02' WITH
         (
         ENDPOINT_URL = 'TCP://COMPUTER02:5022',
         AVAILABILITY_MODE = SYNCHRONOUS_COMMIT,
         FAILOVER_MODE = AUTOMATIC,
```

```
        BACKUP_PRIORITY = 30,
        SECONDARY_ROLE (ALLOW_CONNECTIONS = NO,
           READ_ONLY_ROUTING_URL = 'TCP://COMPUTER02:1433' ),
        PRIMARY_ROLE (ALLOW_CONNECTIONS = READ_WRITE,
           READ_ONLY_ROUTING_LIST = (COMPUTER03) ),
        SESSION_TIMEOUT = 10
        ),

    'COMPUTER03' WITH
        (
        ENDPOINT_URL = 'TCP://COMPUTER03:5022',
        AVAILABILITY_MODE = ASYNCHRONOUS_COMMIT,
        FAILOVER_MODE =  MANUAL,
        BACKUP_PRIORITY = 90,
        SECONDARY_ROLE (ALLOW_CONNECTIONS = READ_ONLY,
           READ_ONLY_ROUTING_URL = 'TCP://COMPUTER03:1433' ),
        PRIMARY_ROLE (ALLOW_CONNECTIONS = READ_WRITE,
           READ_ONLY_ROUTING_LIST = NONE ),
        SESSION_TIMEOUT = 10
        )

LISTENER 'MyAgListenerIvP6' ( WITH IP (
('2001:db88:f0:f00f::cf3c'),('2001:4898:e0:f213::4ce2') ) , PORT = 60173 );
GO
```

Related Tasks

- Create an Availability Group (Transact-SQL)
- Use the New Availability Group Wizard (SQL Server Management Studio)
- Use the New Availability Group Dialog Box (SQL Server Management Studio)
- Use the New Availability Group Wizard (SQL Server Management Studio)

See Also

ALTER AVAILABILITY GROUP (Transact-SQL)

ALTER DATABASE SET HADR (Transact-SQL)

DROP AVAILABILITY GROUP (Transact-SQL)
Troubleshooting AlwaysOn Availability Groups Configuration (SQL Server)
Overview of AlwaysOn Availability Groups (SQL Server)
Client Connectivity and Application Failover (AlwaysOn Availability Groups)

CREATE BROKER PRIORITY

Defines a priority level and the set of criteria for determining which Service Broker conversations to assign the priority level. The priority level is assigned to any conversation endpoint that uses the same combination of contracts and services that are specified in the conversation priority. Priorities range in value from 1 (low) to 10 (high). The default is 5.

 Transact-SQL Syntax Conventions

Syntax

```
CREATE BROKER PRIORITY ConversationPriorityName
FOR CONVERSATION
[ SET ( [ CONTRACT_NAME = {ContractName | ANY } ]
      [ [ , ] LOCAL_SERVICE_NAME = {LocalServiceName | ANY } ]
      [ [ , ] REMOTE_SERVICE_NAME = {'RemoteServiceName' | ANY } ]
      [ [ , ] PRIORITY_LEVEL = {PriorityValue | DEFAULT } ]
     )
]
[;]
```

Arguments

ConversationPriorityName

Specifies the name for this conversation priority. The name must be unique in the current database, and must conform to the rules for Database Engine identifiers.

SET

Specifies the criteria for determining if the conversation priority applies to a conversation. If specified, SET must contain at least one criterion: CONTRACT_NAME, LOCAL_SERVICE_NAME, REMOTE_SERVICE_NAME, or PRIORITY_LEVEL. If SET is not specified, the defaults are set for all three criteria.

CONTRACT_NAME = {ContractName | ANY}

Specifies the name of a contract to be used as a criterion for determining if the conversation priority applies to a conversation. ContractName is a Database Engine identifier, and must specify the name of a contract in the current database.

ContractName

Specifies that the conversation priority can be applied only to conversations where the BEGIN DIALOG statement that started the conversation specified ON CONTRACT ContractName.

ANY

Specifies that the conversation priority can be applied to any conversation, regardless of which contract it uses.

The default is ANY.

LOCAL_SERVICE_NAME = {LocalServiceName | ANY}

Specifies the name of a service to be used as a criterion to determine if the conversation priority applies to a conversation endpoint.

LocalServiceName is a Database Engine identifier. It must specify the name of a service in the current database.

LocalServiceName

Specifies that the conversation priority can be applied to the following:

- Any initiator conversation endpoint whose initiator service name matches LocalServiceName.
- Any target conversation endpoint whose target service name matches LocalServiceName.

ANY

- Specifies that the conversation priority can be applied to any conversation endpoint, regardless of the name of the local service used by the endpoint.

The default is ANY.

REMOTE_SERVICE_NAME = {'RemoteServiceName' | ANY}

Specifies the name of a service to be used as a criterion to determine if the conversation priority applies to a conversation endpoint.

RemoteServiceName is a literal of type **nvarchar(256)**. Service Broker uses a byte-by-byte comparison to match the RemoteServiceName string. The comparison is case-sensitive and does not consider the current collation. The target service can be in the current instance of the Database Engine, or a remote instance of the Database Engine.

'RemoteServiceName'

Specifies that the conversation priority can be applied to the following:

- Any initiator conversation endpoint whose associated target service name matches RemoteServiceName.
- Any target conversation endpoint whose associated initiator service name matches RemoteServiceName.

ANY

Specifies that the conversation priority can be applied to any conversation endpoint, regardless of the name of the remote service associated with the endpoint.

The default is ANY.

PRIORITY_LEVEL = { PriorityValue | DEFAULT }

Specifies the priority to assign any conversation endpoint that use the contracts and services specified in the conversation priority. PriorityValue must be an integer literal from 1 (lowest priority) to 10 (highest priority). The default is 5.

Remarks

Service Broker assigns priority levels to conversation endpoints. The priority levels control the priority of the operations associated with the endpoint. Each conversation has two conversation endpoints:

- The initiator conversation endpoint associates one side of the conversation with the initiator service and initiator queue. The initiator conversation endpoint is created when the BEGIN DIALOG statement is run. The operations associated with the initiator conversation endpoint include:
  - Sends from the initiator service.
  - Receives from the initiator queue.
  - Getting the next conversation group from the initiator queue.
- The target conversation endpoint associates the other side of the conversation with the target service and queue. The target conversation endpoint is created when the conversation is used to send a message to the target queue. The operations associated with the target conversation endpoint include:
  - Receives from the target queue.
  - Sends from the target service.
  - Getting the next conversation group from the target queue.

Service Broker assigns conversation priority levels when conversation endpoints are created. The conversation endpoint retains the priority level until the conversation ends. New priorities or changes to existing priorities are not applied to existing conversations.

Service Broker assigns a conversation endpoint the priority level from the conversation priority whose contract and services criteria best match the properties of the endpoint. The following table shows the match precedence:

Operation contract	Operation local service	Operation remote service
ContractName	LocalServiceName	RemoteServiceName
ContractName	LocalServiceName	ANY
ContractName	ANY	RemoteServiceName

Operation contract	Operation local service	Operation remote service
ContractName	ANY	ANY
ANY	LocalServiceName	RemoteServiceName
ANY	LocalServiceName	ANY
ANY	ANY	RemoteServiceName
ANY	ANY	ANY

Service Broker first looks for a priority whose specified contract, local service, and remote service matches those that the operation uses. If one is not found, Service Broker looks for a priority with a contract and local service that matches those that the operation uses, and where the remote service was specified as ANY. This continues for all the variations that are listed in the precedence table. If no match is found, the operation is assigned the default priority of 5.

Service Broker independently assigns a priority level to each conversation endpoint. To have Service Broker assign priority levels to both the initiator and target conversation endpoints, you must ensure that both endpoints are covered by conversation priorities. If the initiator and target conversation endpoints are in separate databases, you must create conversation priorities in each database. The same priority level is usually specified for both of the conversation endpoints for a conversation, but you can specify different priority levels.

Priority levels are always applied to operations that receive messages or conversation group identifiers from a queue. Priority levels are also applied when transmitting messages from one instance of the Database Engine to another.

Priority levels are not used when transmitting messages:

- From a database where the HONOR_BROKER_PRIORITY database option is set to OFF. For more information, see ALTER DATABASE SET Options (Transact-SQL).
- Between services in the same instance of the Database Engine.
- All Service Broker operations in a database are assigned default priorities of 5 if no conversation priorities have been created in the database.

Permissions

Permission for creating a conversation priority defaults to members of the db_ddladmin or db_owner fixed database roles, and to the sysadmin fixed server role. Requires ALTER permission on the database.

Examples

A. Assigning a priority level to both directions of a conversation.

These two conversation priorities ensure that all operations that use `SimpleContract` between `TargetService` and the `InitiatorAService` are assigned priority level 3.

```
CREATE BROKER PRIORITY InitiatorAToTargetPriority
```

```
    FOR CONVERSATION
    SET (CONTRACT_NAME = SimpleContract,
         LOCAL_SERVICE_NAME = InitiatorServiceA,
         REMOTE_SERVICE_NAME = N'TargetService',
         PRIORITY_LEVEL = 3);
CREATE BROKER PRIORITY TargetToInitiatorAPriority
    FOR CONVERSATION
    SET (CONTRACT_NAME = SimpleContract,
         LOCAL_SERVICE_NAME = TargetService,
         REMOTE_SERVICE_NAME = N'InitiatorServiceA',
         PRIORITY_LEVEL = 3);
```

B. Setting the priority level for all conversations that use a contract

Assigns a priority level of `7` to all operations that use a contract named `SimpleContract`. This assumes that there are no other priorities that specify both `SimpleContract` and either a local or a remote service.

```
CREATE BROKER PRIORITY SimpleContractDefaultPriority
    FOR CONVERSATION
    SET (CONTRACT_NAME = SimpleContract,
         LOCAL_SERVICE_NAME = ANY,
         REMOTE_SERVICE_NAME = ANY,
         PRIORITY_LEVEL = 7);
```

C. Setting a base priority level for a database.

Defines conversation priorities for two specific services, and then defines a conversation priority that will match all other conversation endpoints. This does not replace the default priority, which is always 5, but does minimize the number of items that are assigned the default.

```
CREATE BROKER PRIORITY [//Adventure-Works.com/Expenses/ClaimPriority]
    FOR CONVERSATION
    SET (CONTRACT_NAME = ANY,
         LOCAL_SERVICE_NAME = //Adventure-Works.com/Expenses/ClaimService,
         REMOTE_SERVICE_NAME = ANY,
         PRIORITY_LEVEL = 9);
CREATE BROKER PRIORITY [//Adventure-Works.com/Expenses/ApprovalPriority]
    FOR CONVERSATION
    SET (CONTRACT_NAME = ANY,
```

```
          LOCAL_SERVICE_NAME = //Adventure-Works.com/Expenses/ClaimService,
          REMOTE_SERVICE_NAME = ANY,
          PRIORITY_LEVEL = 6);
CREATE BROKER PRIORITY [//Adventure-Works.com/Expenses/BasePriority]
    FOR CONVERSATION
    SET (CONTRACT_NAME = ANY,
          LOCAL_SERVICE_NAME = ANY,
          REMOTE_SERVICE_NAME = ANY,
          PRIORITY_LEVEL = 3);
```

D. Creating three priority levels for a target service by using services

Supports a system that provides three levels of performance: Gold (high), Silver (medium), and Bronze (low). There is one contract, but each level has a separate initiator service. All initiator services communicate to a central target service.

```
CREATE BROKER PRIORITY GoldInitToTargetPriority
    FOR CONVERSATION
    SET (CONTRACT_NAME = SimpleContract,
          LOCAL_SERVICE_NAME = GoldInitiatorService,
          REMOTE_SERVICE_NAME = N'TargetService',
          PRIORITY_LEVEL = 6);
CREATE BROKER PRIORITY GoldTargetToInitPriority
    FOR CONVERSATION
    SET (CONTRACT_NAME = SimpleContract,
          LOCAL_SERVICE_NAME = TargetService,
          REMOTE_SERVICE_NAME = N'GoldInitiatorService',
          PRIORITY_LEVEL = 6);
CREATE BROKER PRIORITY SilverInitToTargetPriority
    FOR CONVERSATION
    SET (CONTRACT_NAME = SimpleContract,
          LOCAL_SERVICE_NAME = SilverInitiatorService,
          REMOTE_SERVICE_NAME = N'TargetService',
          PRIORITY_LEVEL = 4);
CREATE BROKER PRIORITY SilverTargetToInitPriority
    FOR CONVERSATION
    SET (CONTRACT_NAME = SimpleContract,
```

```
          LOCAL_SERVICE_NAME = TargetService,
          REMOTE_SERVICE_NAME = N'SilverInitiatorService',
          PRIORITY_LEVEL = 4);
CREATE BROKER PRIORITY BronzeInitToTargetPriority
    FOR CONVERSATION
    SET (CONTRACT_NAME = SimpleContract,
          LOCAL_SERVICE_NAME = BronzeInitiatorService,
          REMOTE_SERVICE_NAME = N'TargetService',
          PRIORITY_LEVEL = 2);
CREATE BROKER PRIORITY BronzeTargetToInitPriority
    FOR CONVERSATION
    SET (CONTRACT_NAME = SimpleContract,
          LOCAL_SERVICE_NAME = TargetService,
          REMOTE_SERVICE_NAME = N'BronzeInitiatorService',
          PRIORITY_LEVEL = 2);
```

E. Creating three priority levels for multiple services using contracts

Supports a system that provides three levels of performance: Gold (high), Silver (medium), and Bronze (low). Each level has a separate contract. These priorities apply to any services that are referenced by conversations that use the contracts.

```
CREATE BROKER PRIORITY GoldPriority
    FOR CONVERSATION
    SET (CONTRACT_NAME = GoldContract,
          LOCAL_SERVICE_NAME = ANY,
          REMOTE_SERVICE_NAME = ANY,
          PRIORITY_LEVEL = 6);
CREATE BROKER PRIORITY SilverPriority
    FOR CONVERSATION
    SET (CONTRACT_NAME = SilverContract,
          LOCAL_SERVICE_NAME = ANY,
          REMOTE_SERVICE_NAME = ANY,
          PRIORITY_LEVEL = 4);
CREATE BROKER PRIORITY BronzePriority
    FOR CONVERSATION
    SET (CONTRACT_NAME = BronzeContract,
```

```
    LOCAL_SERVICE_NAME = ANY,
    REMOTE_SERVICE_NAME = ANY,
    PRIORITY_LEVEL = 2);
```

See Also

ALTER BROKER PRIORITY (Transact-SQL)
BEGIN DIALOG CONVERSATION (Transact-SQL)
CREATE CONTRACT (Transact-SQL)
CREATE QUEUE (Transact-SQL)
CREATE SERVICE (Transact-SQL)
DROP BROKER PRIORITY (Transact-SQL)
GET CONVERSATION GROUP (Transact-SQL)
RECEIVE (Transact-SQL)
SEND (Transact-SQL)
sys.conversation_priorities (Transact-SQL)

CREATE CERTIFICATE

Adds a certificate to a database.

 Transact-SQL Syntax Conventions

Syntax

```
CREATE CERTIFICATE certificate_name [ AUTHORIZATION user_name ]
   { FROM <existing_keys> | <generate_new_keys> }
   [ ACTIVE FOR BEGIN_DIALOG = { ON | OFF } ]

<existing_keys> ::=
   ASSEMBLY assembly_name
   |{
      [ EXECUTABLE ] FILE = 'path_to_file'
      [ WITH PRIVATE KEY ( <private_key_options> ) ]
    }
   |{
      BINARY = asn_encoded_certificate
      [ WITH PRIVATE KEY ( <private_key_options> ) ]
    }
```

```
<generate_new_keys> ::=
   [ ENCRYPTION BY PASSWORD = 'password' ]
   WITH SUBJECT = 'certificate_subject_name'
   [ , <date_options> [ ,...n ] ]

<private_key_options> ::=
    {
     FILE = 'path_to_private_key'
      [ , DECRYPTION BY PASSWORD = 'password' ]
      [ , ENCRYPTION BY PASSWORD = 'password' ]
    }
   |
    {
     BINARY = private_key_bits
      [ , DECRYPTION BY PASSWORD = 'password' ]
      [ , ENCRYPTION BY PASSWORD = 'password' ]
    }

<date_options> ::=
   START_DATE = 'datetime' | EXPIRY_DATE = 'datetime'
```

Arguments

certificate_name

Is the name by which the certificate will be known in the database.

AUTHORIZATION user_name

Is the name of the user that will own this certificate.

ASSEMBLY assembly_name

Specifies a signed assembly that has already been loaded into the database.

[EXECUTABLE] FILE = 'path_to_file'

Specifies the complete path, including file name, to a DER-encoded file that contains the certificate. If the EXECUTABLE option is used, the file is a DLL that has been signed by the certificate. path_to_file can be a local path or a UNC path to a network location. The file will be accessed in the security context of the SQL Server service account. This account must have the required file-system permissions.

WITH PRIVATE KEY

Specifies that the private key of the certificate is loaded into SQL Server. This clause is only

valid when the certificate is being created from a file. To load the private key of an assembly, use ALTER CERTIFICATE.

FILE = 'path_to_private_key'

Specifies the complete path, including file name, to the private key. path_to_private_key can be a local path or a UNC path to a network location. The file will be accessed in the security context of the SQL Server service account. This account must have the necessary file-system permissions.

Note

This option is not available in a contained database.

asn_encoded_certificate

ASN encoded certificate bits specified as a binary constant.

BINARY = private_key_bits

Private key bits specified as binary constant. These bits can be in encrypted form. If encrypted, the user must provide a decryption password. Password policy checks are not performed on this password. The private key bits should be in a PVK file format.

DECRYPTION BY PASSWORD = 'key_password'

Specifies the password required to decrypt a private key that is retrieved from a file. This clause is optional if the private key is protected by a null password. Saving a private key to a file without password protection is not recommended. If a password is required but no password is specified, the statement will fail.

ENCRYPTION BY PASSWORD = 'password'

Specifies the password that will be used to encrypt the private key. Use this option only if you want to encrypt the certificate with a password. If this clause is omitted, the private key will be encrypted using the database master key. password must meet the Windows password policy requirements of the computer that is running the instance of SQL Server. For more information, see EVENTDATA (Transact-SQL).

SUBJECT = 'certificate_subject_name'

The term *subject* refers to a field in the metadata of the certificate as defined in the X.509 standard. The subject can be up to 128 characters long. Subjects that exceed 128 characters will be truncated when they are stored in the catalog, but the binary large object (BLOB) that contains the certificate will retain the full subject name.

START_DATE = 'datetime'

Is the date on which the certificate becomes valid. If not specified, START_DATE will be set equal to the current date. START_DATE is in UTC time and can be specified in any format that can be converted to a date and time.

EXPIRY_DATE = 'datetime'

Is the date on which the certificate expires. If not specified, EXPIRY_DATE will be set to a date

one year after START_DATE. EXPIRY_DATE is in UTC time and can be specified in any format that can be converted to a date and time. SQL Server Service Broker checks the expiration date; however, expiration is not enforced when the certificate is used for encryption.

ACTIVE FOR BEGIN_DIALOG = { ON | OFF }

Makes the certificate available to the initiator of a Service Broker dialog conversation. The default value is ON.

Remarks

A certificate is a database-level securable that follows the X.509 standard and supports X.509 V1 fields. CREATE CERTIFICATE can load a certificate from a file or assembly. This statement can also generate a key pair and create a self-signed certificate.

Private keys generated by SQL Server are 1024 bits long. Private keys imported from an external source have a minimum length of 384 bits and a maximum length of 4,096 bits. The length of an imported private key must be an integer multiple of 64 bits.

The private key must correspond to the public key specified by certificate_name.

When you create a certificate from a container, loading the private key is optional. But when SQL Server generates a self-signed certificate, the private key is always created. By default, the private key is encrypted using the database master key. If the database master key does not exist and no password is specified, the statement will fail.

The ENCRYPTION BY PASSWORD option is not required when the private key will be encrypted with the database master key. Use this option only when the private key will be encrypted with a password. If no password is specified, the private key of the certificate will be encrypted using the database master key. Omitting this clause will cause an error if the master key of the database cannot be opened.

You do not have to specify a decryption password when the private key is encrypted with the database master key.

Note

Built-in functions for encryption and signing do not check the expiration dates of certificates. Users of these functions must decide when to check certificate expiration.

A binary description of a certificate can be created by using the CERTENCODED (Transact-SQL) and CERTPRIVATEKEY (Transact-SQL) functions. For an example that uses **CERTPRIVATEKEY** and **CERTENCODED** to copy a certificate to another database, see example B in the topic CERTENCODED (Transact-SQL).

Permissions

Requires CREATE CERTIFICATE permission on the database. Only Windows logins, SQL Server logins, and application roles can own certificates. Groups and roles cannot own certificates.

Examples

A. Creating a self-signed certificate

The following example creates a certificate called `Shipping04`. The private key of this certificate is protected using a password.

```
USE AdventureWorks2012;
CREATE CERTIFICATE Shipping04
   ENCRYPTION BY PASSWORD = 'pGFD4bb925DGvbd2439587y'
   WITH SUBJECT = 'Sammamish Shipping Records',
   EXPIRY_DATE = '20121031';
GO
```

B. Creating a certificate from a file

The following example creates a certificate in the database, loading the key pair from files.

```
USE AdventureWorks2012;
CREATE CERTIFICATE Shipping11
    FROM FILE = 'c:\Shipping\Certs\Shipping11.cer'
    WITH PRIVATE KEY (FILE = 'c:\Shipping\Certs\Shipping11.pvk',
    DECRYPTION BY PASSWORD = 'sldkflk34et6gs%53#v00');
GO
```

C. Creating a certificate from a signed executable file

```
USE AdventureWorks2012;
CREATE CERTIFICATE Shipping19
    FROM EXECUTABLE FILE = 'c:\Shipping\Certs\Shipping19.dll';
GO
```

Alternatively, you can create an assembly from the `dll` file, and then create a certificate from the assembly.

```
USE AdventureWorks2012;
CREATE ASSEMBLY Shipping19
    FROM ' c:\Shipping\Certs\Shipping19.dll'
    WITH PERMISSION_SET = SAFE;
GO
CREATE CERTIFICATE Shipping19 FROM ASSEMBLY Shipping19;
GO
```

See Also

ALTER CERTIFICATE (Transact-SQL)

DROP CERTIFICATE (Transact-SQL)
BACKUP CERTIFICATE (Transact-SQL)
Encryption Hierarchy
EVENTDATA (Transact-SQL)
CERTENCODED (Transact-SQL)
CERTPRIVATEKEY (Transact-SQL)

CREATE COLUMNSTORE INDEX

Creates a columnstore index on a specified table. An *xVelocity memory optimized columnstore index*, is a type of compressed non-clustered index. There is a limit of one columnstore index per table. An index can be created before there is data in the table. A table with a columnstore index cannot be updated. For information about using columnstore indexes, see Columnstore Indexes.

Note
For information about how to create a relational index, see CREATE INDEX (Transact-SQL). For information about how to create an XML index, see CREATE XML INDEX (Transact-SQL). For information about how to create a spatial index, see CREATE SPATIAL INDEX (Transact-SQL).

Transact-SQL Syntax Conventions

Syntax

```
CREATE [ NONCLUSTERED ] COLUMNSTORE INDEX index_name
   ON <object> ( column  [ ,...n ] )
   [ WITH ( <column_index_option> [ ,...n ] ) ]
   [ ON {
        { partition_scheme_name ( column_name ) }
        | filegroup_name
        | "default"
      }
   ]
[ ; ]

<object> ::=
{
   [database_name. [schema_name ] . | schema_name . ]
    table_name
{
```

<column_index_option> ::=

```
{
    DROP_EXISTING = { ON | OFF }
  | MAXDOP = max_degree_of_parallelism
}
```

Arguments

NONCLUSTERED

Creates a columnstore index that specifies the logical ordering of a table. Clustered columnstore indexes are not supported.

COLUMNSTORE

Indicates the index will be a columnstore index.

index_name

Is the name of the index. Index names must be unique within a table or view but do not have to be unique within a database. Index names must follow the rules of identifiers.

column

Is the column or columns on which the index is based. A columnstore index is limited to 1024 columns.

ON partition_scheme_name (column_name)

Specifies the partition scheme that defines the filegroups onto which the partitions of a partitioned index will be mapped. The partition scheme must exist within the database by executing CREATE PARTITION SCHEME. column_name specifies the column against which a partitioned index will be partitioned. This column must match the data type, length, and precision of the argument of the partition function that partition_scheme_name is using. column_name is not restricted to the columns in the index definition. When partitioning a columnstore index, Database Engine adds the partitioning column as a column of the index, if it is not already specified.

If partition_scheme_name or filegroup is not specified and the table is partitioned, the index is placed in the same partition scheme, using the same partitioning column, as the underlying table.

For more information about partitioning indexes, see Partitioned Tables and Indexes.

ON filegroup_name

Creates the specified index on the specified filegroup. If no location is specified and the table or view is not partitioned, the index uses the same filegroup as the underlying table or view. The filegroup must already exist.

ON "default"

Creates the specified index on the default filegroup.

The term default, in this context, is not a keyword. It is an identifier for the default filegroup and must be delimited, as in ON **"**default**"** or ON **[**default**]**. If "default" is specified, the QUOTED_IDENTIFIER option must be ON for the current session. This is the default setting. For more information, see SET QUOTED_IDENTIFIER (Transact-SQL).

<object>::=

Is the fully qualified or nonfully qualified object to be indexed.

database_name

Is the name of the database.

schema_name

Is the name of the schema to which the table belongs.

table_name

Is the name of the table to be indexed.

<column_index_option>::=

Specifies the options to use when you create the column store index.

DROP_EXISTING

Specifies that the named, preexisting index is dropped and rebuilt. The default is OFF.

ON

The existing index is dropped and rebuilt. The index name specified must be the same as a currently existing index; however, the index definition can be modified. For example, you can specify different columns, or index options.

OFF

An error is displayed if the specified index name already exists. The index type cannot be changed by using DROP_EXISTING. In backward compatible syntax, WITH DROP_EXISTING is equivalent to WITH DROP_EXISTING = ON.

MAXDOP = max_degree_of_parallelism

Overrides the Configure the max degree of parallelism Server Configuration Option configuration option for the duration of the index operation. Use MAXDOP to limit the number of processors used in a parallel plan execution. The maximum is 64 processors.

max_degree_of_parallelism can be:

1

Suppresses parallel plan generation.

>1

Restricts the maximum number of processors used in a parallel index operation to the specified number or fewer based on the current system workload.

0 (default)

Uses the actual number of processors or fewer based on the current system workload.

For more information, see Configuring Parallel Index Operations.

Note

Parallel index operations are not available in every edition of Microsoft SQL Server. For a list of features that are supported by the editions of SQL Server, see Features Supported by the Editions of SQL Server 2012.

Remarks

Indexes can be created on a temporary table. When the table is dropped or the session ends, the indexes are dropped.

The common business data types can be included in a columnstore index. The following data types can be included in a columnstore index.

- **char** and **varchar**
- **nchar** and **nvarchar** (except **varchar(max)** and **nvarchar(max)**)
- **decimal** (and **numeric**) (Except with precision greater than 18 digits.)
- **int**, **bigint**, **smallint**, and **tinyint**
- **float** (and **real**)
- **bit**
- **money** and **smallmoney**
- All date and time data types (except **datetimeoffset** with scale greater than 2)

The following data types cannot be included in a columnstore index.

- **binary** and **varbinary**
- **ntext**, **text**, and **image**
- **varchar(max)** and **nvarchar(max)**
- **uniqueidentifier**
- **rowversion** (and **timestamp**)
- **sql_variant**
- **decimal** (and **numeric**) with precision greater than 18 digits
- **datetimeoffset** with scale greater than 2
- CLR types (**hierarchyid** and spatial types)
- **xml**

Basic Restrictions

A columnstore index:

- Cannot have more than 1024 columns.
- Cannot be clustered. Only nonclustered columnstore indexes are available.

- Cannot be a unique index.
- Cannot be created on a view or indexed view.
- Cannot include a sparse column.
- Cannot act as a primary key or a foreign key.
- Cannot be changed using the **ALTER INDEX** statement. Drop and re-create the columnstore index instead. (You can use **ALTER INDEX** to disable and rebuild a columnstore index.)
- Cannot be created by with the **INCLUDE** keyword.
- Cannot include the **ASC** or **DESC** keywords for sorting the index. Columnstore indexes are ordered according to the compression algorithms. Sorting would eliminate many of the performance benefits.

Columnstore indexes cannot be combined with the following features:

- Page and row compression, and **vardecimal** storage format (A columnstore index is already compressed in a different format.)
- Replication
- Change tracking
- Change data capture
- Filestream

For information about the performance benefits and limitations of columnstore indexes, see Understanding Columnstore Indexes.

Permissions

Requires ALTER permission on the table.

Examples

A. Creating a simple nonclustered index

The following example creates a simple table and clustered index, and then demonstrates the syntax of creating a columnstore index.

```
CREATE TABLE SimpleTable
(ProductKey [int] NOT NULL,
OrderDateKey [int] NOT NULL,
DueDateKey [int] NOT NULL,
ShipDateKey [int] NOT NULL);
GO
CREATE CLUSTERED INDEX cl_simple ON SimpleTable (ProductKey);
GO
CREATE NONCLUSTERED COLUMNSTORE INDEX csindx_simple
ON SimpleTable
```

```
(OrderDateKey, DueDateKey, ShipDateKey);
GO
```

B. Creating a simple nonclustered index using all options

The following example creates a simple table and clustered index, and then demonstrates the syntax of creating a columnstore index.

```
CREATE NONCLUSTERED COLUMNSTORE INDEX csindx_simple
ON SimpleTable
(OrderDateKey, DueDateKey, ShipDateKey)
WITH (DROP_EXISTING =  ON,
    MAXDOP = 2)
ON "default"
GO
```

For a more complex example using partitioned tables, see Understanding Columnstore Indexes.

See Also

Understanding Columnstore Indexes
Columnstore Indexes
sys.column_store_dictionaries (Transact-SQL)
sys.column_store_segments (Transact-SQL)
ALTER INDEX (Transact-SQL)
CREATE PARTITION FUNCTION
CREATE PARTITION SCHEME
DROP INDEX
sys.indexes
sys.index_columns

CREATE CONTRACT

Creates a new contract. A contract defines the message types that are used in a Service Broker conversation and also determines which side of the conversation can send messages of that type. Each conversation follows a contract. The initiating service specifies the contract for the conversation when the conversation starts. The target service specifies the contracts that the target service accepts conversations for.

 Transact-SQL Syntax Conventions

Syntax

```
CREATE CONTRACT contract_name
   [ AUTHORIZATION owner_name ]
     ( { { message_type_name | [ DEFAULT ] }
         SENT BY { INITIATOR | TARGET | ANY }
     } [ ,...n] )
[ ; ]
```

Arguments

contract_name

Is the name of the contract to create. A new contract is created in the current database and owned by the principal specified in the AUTHORIZATION clause. Server, database, and schema names cannot be specified. The contract_name can be up to 128 characters.

Note

Do not create a contract that uses the keyword ANY for the contract_name. When you specify ANY for a contract name in CREATE BROKER PRIORITY, the priority is considered for all contracts. It is not limited to a contract whose name is ANY.

AUTHORIZATION owner_name

Sets the owner of the contract to the specified database user or role. When the current user is **dbo** or **sa**, owner_name can be the name of any valid user or role. Otherwise, owner_name must be the name of the current user, the name of a user that the current user has impersonate permissions for, or the name of a role to which the current user belongs. When this clause is omitted, the contract belongs to the current user.

message_type_name

Is the name of a message type to be included as part of the contract.

SENT BY

Specifies which endpoint can send a message of the indicated message type. Contracts document the messages that services can use to have specific conversations. Each conversation has two endpoints: the *initiator* endpoint, the service that started the conversation, and the *target* endpoint, the service that the initiator is contacting.

INITIATOR

Indicates that only the initiator of the conversation can send messages of the specified message type. A service that starts a conversation is referred to as the *initiator* of the conversation.

TARGET

Indicates that only the target of the conversation can send messages of the specified message type. A service that accepts a conversation that was started by another service is

referred to as the *target* of the conversation.

ANY

Indicates that messages of this type can be sent by both the initiator and the target.

[DEFAULT]

Indicates that this contract supports messages of the default message type. By default, all databases contain a message type named DEFAULT. This message type uses a validation of NONE. In the context of this clause, DEFAULT is not a keyword, and must be delimited as an identifier. Microsoft SQL Server also provides a DEFAULT contract which specifies the DEFAULT message type.

Remarks

The order of message types in the contract is not significant. After the target has received the first message, Service Broker allows either side of the conversation to send any message allowed for that side of the conversation at any time. For example, if the initiator of the conversation can send the message type **//Adventure-Works.com/Expenses/SubmitExpense**, Service Broker allows the initiator to send any number of **SubmitExpense** messages during the conversation.

The message types and directions in a contract cannot be changed. To change the AUTHORIZATION for a contract, use the ALTER AUTHORIZATION statement.

A contract must allow the initiator to send a message. The CREATE CONTRACT statement fails when the contract does not contain at least one message type that is SENT BY ANY or SENT BY INITIATOR.

Regardless of the contract, a service can always receive the message types **http://schemas.microsoft.com/SQL/ServiceBroker/DialogTimer**, **http://schemas.microsoft.com/SQL/ServiceBroker/Error**, and **http://schemas.microsoft.com/SQL/ServiceBroker/EndDialog**. Service Broker uses these message types for system messages to the application.

A contract cannot be a temporary object. Contract names starting with # are permitted, but are permanent objects.

Permissions

By default, members of the **db_ddladmin** or **db_owner** fixed database roles and the **sysadmin** fixed server role can create contracts.

By default, the owner of the contract, members of the **db_ddladmin** or **db_owner** fixed database roles, and members of the **sysadmin** fixed server role have REFERENCES permission on a contract.

The user executing the CREATE CONTRACT statement must have REFERENCES permission on all message types specified.

Examples

A. Creating a contract

The following example creates an expense reimbursement contract based on three message types.

```
CREATE MESSAGE TYPE
    [//Adventure-Works.com/Expenses/SubmitExpense]
    VALIDATION = WELL_FORMED_XML ;

CREATE MESSAGE TYPE
    [//Adventure-Works.com/Expenses/ExpenseApprovedOrDenied]
    VALIDATION = WELL_FORMED_XML ;

CREATE MESSAGE TYPE
    [//Adventure-Works.com/Expenses/ExpenseReimbursed]
    VALIDATION= WELL_FORMED_XML ;

CREATE CONTRACT
    [//Adventure-Works.com/Expenses/ExpenseSubmission]
    ( [//Adventure-Works.com/Expenses/SubmitExpense]
          SENT BY INITIATOR,
      [//Adventure-Works.com/Expenses/ExpenseApprovedOrDenied]
          SENT BY TARGET,
      [//Adventure-Works.com/Expenses/ExpenseReimbursed]
          SENT BY TARGET
    ) ;
```

See Also

DROP CONTRACT
EVENTDATA

CREATE CREDENTIAL

Creates a credential.

 Transact-SQL Syntax Conventions

Syntax

CREATE CREDENTIAL **credential_name** WITH IDENTITY = '**identity_name**'

[, SECRET = '**secret**']

[FOR CRYPTOGRAPHIC PROVIDER **cryptographic_provider_name**]

Arguments

credential_name

Specifies the name of the credential being created. credential_name cannot start with the number (#) sign. System credentials start with ##.

IDENTITY = 'identity_name'

Specifies the name of the account to be used when connecting outside the server.

SECRET = 'secret'

Specifies the secret required for outgoing authentication. This clause is optional.

FOR CRYPTOGRAPHIC PROVIDER cryptographic_provider_name

Specifies the name of an *Enterprise Key Management Provider (EKM)*. For more information about Key Management, see Understanding Extensible Key Management (EKM).

Remarks

A credential is a record that contains the authentication information that is required to connect to a resource outside SQL Server. Most credentials include a Windows user and password.

When IDENTITY is a Windows user, the secret can be the password. The secret is encrypted using the service master key. If the service master key is regenerated, the secret is re-encrypted using the new service master key.

After creating a credential, you can map it to a SQL Server login by using CREATE LOGIN or ALTER LOGIN. A SQL Server login can be mapped to only one credential, but a single credential can be mapped to multiple SQL Server logins. For more information, see sys.credentials (Transact-SQL).

Information about credentials is visible in the sys.credentials catalog view.

If there is no login mapped credential for the provider, the credential mapped to SQL Server service account is used.

A login can have multiple credentials mapped to it as long as they are used with distinctive providers. There must be only one mapped credential per provider per login. The same credential can be mapped to other logins.

Permissions

Requires **ALTER ANY CREDENTIAL** permission.

Examples

The following example creates the credential called `AlterEgo`. The credential contains the Windows user `Mary5` and a password.

```
CREATE CREDENTIAL AlterEgo WITH IDENTITY = 'Mary5',
```

```
    SECRET = '<EnterStrongPasswordHere>';
GO
```

The following example uses a previously created account called `User1OnEKM` on an EKM module through the EKM's Management tools, with a basic account type and password. The **sysadmin** account on the server creates a credential that is used to connect to the EKM account, and assigns it to the `User1` SQL Server account:

```
CREATE CREDENTIAL CredentialForEKM
WITH IDENTITY='User1OnEKM'
, SECRET='<EnterStrongPasswordHere>'
  FOR CRYPTOGRAPHIC PROVIDER MyEKMProvider;
GO
/* Modify the login to assign the cryptographic provider credential */
ALTER LOGIN User1
ADD CREDENTIAL CredentialForEKM;
/* Modify the login to assign a non cryptographic provider credential */
ALTER LOGIN User1
WITH CREDENTIAL = AlterEgo;
GO
```

See Also

Credentials
ALTER CREDENTIAL (Transact-SQL)
DROP CREDENTIAL (Transact-SQL)
CREATE LOGIN (Transact-SQL)
ALTER LOGIN (Transact-SQL)
sys.credentials (Transact-SQL)

CREATE CRYPTOGRAPHIC PROVIDER

Creates a cryptographic provider within SQL Server from an Extensible Key Management (EKM) provider.

 Transact-SQL Syntax Conventions

Syntax

CREATE CRYPTOGRAPHIC PROVIDER **`provider_name`**
  FROM FILE = **`path_of_DLL`**

Arguments

provider_name

Is the name of the Extensible Key Management provider.

path_of_DLL

Is the path of the .dll file that implements the SQL Server Extensible Key Management interface.

Remarks

All keys created by a provider will reference the provider by its GUID. The GUID is retained across all versions of the DLL.

The DLL that implements SQLEKM interface must be digitally signed using any certificate. SQL Server will verify the signature. This includes its certificate chain, which must have its root installed at the **Trusted Root Cert Authorities** location on a Windows system. If the signature is not verified correctly, the CREATE CRYPTOGRAPHIC PROVIDER statement will fail. For more information about certificates and certificate chains, see SQL Server Certificates and Asymmetric Keys.

When an EKM provider dll does not implement all of the necessary methods, CREATE CRYPTOGRAPHIC PROVIDER can return error 33085:

```
One or more methods cannot be found in cryptographic provider library
'%.*ls'.
```

When the header file used to create the EKM provider dll is out of date, CREATE CRYPTOGRAPHIC PROVIDER can return error 33032:

```
SQL Crypto API version '%02d.%02d' implemented by provider is not supported.
Supported version is '%02d.%02d'.
```

Permissions

Requires CONTROL permission on the symmetric key.

Examples

The following example creates a cryptographic provider called `SecurityProvider` in SQL Server from a .dll file. The .dll file is named `c:\SecurityProvider\SecurityProvider_v1.dll` and it is installed on the server. The provider's certificate must first be installed on the server.

```
-- Install the provider
CREATE CRYPTOGRAPHIC PROVIDER SecurityProvider
    FROM FILE = 'c:\SecurityProvider\SecurityProvider_v1.dll'
```

See Also

Understanding Extensible Key Management (EKM)

ALTER CRYPTOGRAPHIC PROVIDER (Transact-SQL)

DROP CRYPTOGRAPHIC PROVIDER (Transact-SQL)

CREATE SYMMETRIC KEY (Transact-SQL)

CREATE DATABASE

Creates a new database and the files used to store the database, a database snapshot, or attaches a database from the detached files of a previously created database.

 Transact-SQL Syntax Conventions

Syntax

```
CREATE DATABASE database_name
[ CONTAINMENT = { NONE | PARTIAL } ]
[ ON
    [ PRIMARY ] <filespec> [ ,...n ]
    [ , <filegroup> [ ,...n ] ]
    [ LOG ON <filespec> [ ,...n ] ]
]
[ COLLATE collation_name ]
[ WITH  <option> [,...n ] ]
[;]

<option> ::=
{
    FILESTREAM ( <filestream_option> [,...n ] )
  | DEFAULT_FULLTEXT_LANGUAGE = { lcid | language_name | language_alias }
  | DEFAULT_LANGUAGE = { lcid | language_name | language_alias }
  | NESTED_TRIGGERS = { OFF | ON }
  | TRANSFORM_NOISE_WORDS = { OFF | ON}
  | TWO_DIGIT_YEAR_CUTOFF = <two_digit_year_cutoff>
  | DB_CHAINING { OFF | ON }
  | TRUSTWORTHY { OFF | ON }
}

<filestream_option> ::=
{
    NON_TRANSACTED_ACCESS = { OFF | READ_ONLY | FULL }
  | DIRECTORY_NAME = 'directory_name'
```

}

To attach a database

CREATE DATABASE **database_name**
  ON <filespec> [,...**n**]
  FOR { { ATTACH [WITH <attach_database_option> [, ...**n**]] }
    | ATTACH_REBUILD_LOG }
[;]

<filespec> ::=

{
(
  NAME = **logical_file_name ,**
  FILENAME = { **'os_file_name'** | **'filestream_path'** }
  [, SIZE = **size** [KB | MB | GB | TB]]
  [, MAXSIZE = { **max_size** [KB | MB | GB | TB] | UNLIMITED }]
  [, FILEGROWTH = **growth_increment** [KB | MB | GB | TB | %]]
)
}

<filegroup> ::=

{
FILEGROUP **filegroup_name** [CONTAINS FILESTREAM] [DEFAULT]
  <filespec> [,...**n**]
}

<attach_database_option> ::=

{
    <service_broker_option>
  | RESTRICTED_USER
  | FILESTREAM (DIRECTORY_NAME = { **'directory_name'** | NULL })
}

<service_broker_option> ::=

{

```
    ENABLE_BROKER
  | NEW_BROKER
  | ERROR_BROKER_CONVERSATIONS
}
```

Create a database snapshot

```
CREATE DATABASE database_snapshot_name
    ON
    (
      NAME = logical_file_name,
      FILENAME = 'os_file_name'
    ) [ ,...n ]
    AS SNAPSHOT OF source_database_name
[;]
```

Arguments

database_name

Is the name of the new database. Database names must be unique within an instance of SQL Server and comply with the rules for identifiers.

database_name can be a maximum of 128 characters, unless a logical name is not specified for the log file. If a logical log file name is not specified, SQL Server generates the logical_file_name and the os_file_name for the log by appending a suffix to database_name. This limits database_name to 123 characters so that the generated logical file name is no more than 128 characters.

If data file name is not specified, SQL Server uses database_name as both the logical_file_name and as the os_file_name. The default path is obtained from the registry. The default path can be changed by using the **Server Properties (Database Settings Page)** in Management Studio. Changing the default path requires restarting SQL Server.

CONTAINMENT

Specifies the containment status of the database. NONE = non-contained database. PARTIAL = partially contained database.

ON

Specifies that the disk files used to store the data sections of the database, data files, are explicitly defined. ON is required when followed by a comma-separated list of <filespec> items that define the data files for the primary filegroup. The list of files in the primary filegroup can be followed by an optional, comma-separated list of <filegroup> items that define user filegroups and their files.

PRIMARY

Specifies that the associated <filespec> list defines the primary file. The first file specified in the <filespec> entry in the primary filegroup becomes the primary file. A database can have only one primary file. For more information, see Database Files and Filegroups.

If PRIMARY is not specified, the first file listed in the CREATE DATABASE statement becomes the primary file.

LOG ON

Specifies that the disk files used to store the database log, log files, are explicitly defined. LOG ON is followed by a comma-separated list of <filespec> items that define the log files. If LOG ON is not specified, one log file is automatically created, which has a size that is 25 percent of the sum of the sizes of all the data files for the database, or 512 KB, whichever is larger. This file is placed in the default log-file location. For information about this location, see How to: View or Change the Default Locations for Database Files (SQL Server Management Studio).

LOG ON cannot be specified on a database snapshot.

COLLATE collation_name

Specifies the default collation for the database. Collation name can be either a Windows collation name or a SQL collation name. If not specified, the database is assigned the default collation of the instance of SQL Server. A collation name cannot be specified on a database snapshot.

A collation name cannot be specified with the FOR ATTACH or FOR ATTACH_REBUILD_LOG clauses. For information about how to change the collation of an attached database, visit this Microsoft Web site.

For more information about the Windows and SQL collation names, see COLLATE (Transact-SQL).

Note

Contained databases are collated differently than non-contained databases. Please see Contained Database Collations for more information.

WITH <option>

The following options are allowable only when CONTAINMENT has been set to PARTIAL. If CONTAINMENT is set to NONE, errors will occur.

- **<filestream_options>**

  NON_TRANSACTED_ACCESS = { OFF | READ_ONLY | FULL }

  Specifies the level of non-transactional FILESTREAM access to the database.

Value	Description
OFF	Non-transactional access is disabled.

READONLY	FILESTREAM data in this database can be read by non-transactional processes.
FULL	Full non-transactional access to FILESTREAM FileTables is enabled.

DIRECTORY_NAME = <directory_name>

A windows-compatible directory name. This name should be unique among all the Database_Directory names in the SQL Server instance. Uniqueness comparison is case-insensitive, regardless of SQL Server collation settings. This option should be set before creating a FileTable in this database.

- **DEFAULT_FULLTEXT_LANGUAGE = <lcid> | <language name> | <language alias>**

  See Configure the default full-text language Server Configuration Option for a full description of this option.

- **DEFAULT_LANGUAGE = <lcid> | <language name> | <language alias>**

  See Configure the default language Server Configuration Option for a full description of this option.

- **NESTED_TRIGGERS = { OFF | ON}**

  See Configure the nested triggers Server Configuration Option for a full description of this option.

- **TRANSFORM_NOISE_WORDS = { OFF | ON}**

  See transform noise words Option for a full description of this option.

- **TWO_DIGIT_YEAR_CUTOFF = { 2049 | <any year between 1753 and 9999> }**

  Four digits representing a year. 2049 is the default value. See Configure the two digit year cutoff Server Configuration Option for a full description of this option.

- **DB_CHAINING { OFF | ON }**

  When ON is specified, the database can be the source or target of a cross-database ownership chain.

  When OFF, the database cannot participate in cross-database ownership chaining. The default is OFF.

  Important

  The instance of SQL Server will recognize this setting when the cross db ownership chaining server option is 0 (OFF). When cross db ownership chaining is 1 (ON), all user databases can participate in cross-database ownership chains, regardless of the value of this option. This option is set by using sp_configure.

  To set this option, requires membership in the sysadmin fixed server role. The DB_CHAINING option cannot be set on these system databases: master, model, tempdb.

- **TRUSTWORTHY { OFF | ON }**

When ON is specified, database modules (for example, views, user-defined functions, or stored procedures) that use an impersonation context can access resources outside the database.

When OFF, database modules in an impersonation context cannot access resources outside the database. The default is OFF.

TRUSTWORTHY is set to OFF whenever the database is attached.

By default, all system databases except the msdb database have TRUSTWORTHY set to OFF. The value cannot be changed for the model and tempdb databases. We recommend that you never set the TRUSTWORTHY option to ON for the master database.

To set this option, requires membership in the sysadmin fixed server role.

FOR ATTACH [WITH < attach_database_option >]

Specifies that the database is created by attaching an existing set of operating system files. There must be a <filespec> entry that specifies the primary file. The only other <filespec> entries required are those for any files that have a different path from when the database was first created or last attached. A <filespec> entry must be specified for these files.

FOR ATTACH requires the following:

- All data files (MDF and NDF) must be available.
- If multiple log files exist, they must all be available.

If a read/write database has a single log file that is currently unavailable, and if the database was shut down with no users or open transactions before the attach operation, FOR ATTACH automatically rebuilds the log file and updates the primary file. In contrast, for a read-only database, the log cannot be rebuilt because the primary file cannot be updated. Therefore, when you attach a read-only database with a log that is unavailable, you must provide the log files, or the files in the FOR ATTACH clause.

Note

A database created by a more recent version of SQL Server cannot be attached in earlier versions. The source database must be at least version 90 (SQL Server 2005) to attach to SQL Server 2012. SQL Server 2005 databases that have a compatibility level less than 90 will be set to compatibility 90 when they are attached.

In SQL Server, any full-text files that are part of the database that is being attached will be attached with the database. To specify a new path of the full-text catalog, specify the new location without the full-text operating system file name. For more information, see the Examples section.

Attaching a database that contains a FILESTREAM option of "Directory name", into a SQL Server instance will prompt SQL Server to verify that the Database_Directory name is unique. If it is not, the attach operation fails with the error, "FILESTREAM Database_Directory name <name> is not unique in this SQL Server instance". To avoid this error, the optional parameter, directory_name, should be passed in to this operation.

FOR ATTACH cannot be specified on a database snapshot.

FOR ATTACH can specify the RESTRICTED_USER option. RESTRICTED_USER allows for only members of the db_owner fixed database role and dbcreator and sysadmin fixed server roles to connect to the database, but does not limit their number. Attempts by unqualified users are refused.

If the database uses Service Broker, use the WITH <service_broker_option> in your FOR ATTACH clause:

<service_broker_option>

Controls Service Broker message delivery and the Service Broker identifier for the database. Service Broker options can only be specified when the FOR ATTACH clause is used.

ENABLE_BROKER

Specifies that Service Broker is enabled for the specified database. That is, message delivery is started, and is_broker_enabled is set to true in the sys.databases catalog view. The database retains the existing Service Broker identifier.

NEW_BROKER

Creates a new service_broker_guid value in both sys.databases and the restored database and ends all conversation endpoints with clean up. The broker is enabled, but no message is sent to the remote conversation endpoints. Any route that references the old Service Broker identifier must be re-created with the new identifier.

ERROR_BROKER_CONVERSATIONS

Ends all conversations with an error stating that the database is attached or restored. The broker is disabled until this operation is completed and then enabled. The database retains the existing Service Broker identifier.

When you attach a replicated database that was copied instead of being detached, consider the following:

- If you attach the database to the same server instance and version as the original database, no additional steps are required.
- If you attach the database to the same server instance but with an upgraded version, you must execute sp_vupgrade_replication to upgrade replication after the attach operation is complete.
- If you attach the database to a different server instance, regardless of version, you must execute sp_removedbreplication to remove replication after the attach operation is complete.

Note

Attach works with the **vardecimal** storage format, but the SQL Server Database Engine must be upgraded to at least SQL Server 2005 Service Pack 2. You cannot attach a database using vardecimal storage format to an earlier version of SQL Server. For more information about the **vardecimal** storage format, see Data Compression.

When a database is first attached or restored to a new instance of SQL Server, a copy of the database master key (encrypted by the service master key) is not yet stored in the server. You must use the **OPEN MASTER KEY** statement to decrypt the database master key (DMK). Once the DMK has been decrypted, you have the option of enabling automatic decryption in the future by using the **ALTER MASTER KEY REGENERATE** statement to provision the server with a copy of the DMK, encrypted with the service master key (SMK). When a database has been upgraded from an earlier version, the DMK should be regenerated to use the newer AES algorithm. For more information about regenerating the DMK, see ALTER MASTER KEY (Transact-SQL). The time required to regenerate the DMK key to upgrade to AES depends upon the number of objects protected by the DMK. Regenerating the DMK key to upgrade to AES is only necessary once, and has no impact on future regenerations as part of a key rotation strategy. For information about how to upgrade a database by using attach, see How to: Upgrade a Database Using Detach and Attach (Transact-SQL).

Security Note We recommend that you do not attach databases from unknown or untrusted sources. Such databases could contain malicious code that might execute unintended Transact-SQL code or cause errors by modifying the schema or the physical database structure. Before you use a database from an unknown or untrusted source, run DBCC CHECKDB on the database on a nonprocession server, and also examine the code, such as stored procedures or other user-defined code, in the database.

Note

The **TRUSTWORTHY** and **DB_CHAINING** options have no affect when attaching a database.

FOR ATTACH_REBUILD_LOG

Specifies that the database is created by attaching an existing set of operating system files. This option is limited to read/write databases. There must be a <*filespec*> entry specifying the primary file. If one or more transaction log files are missing, the log file is rebuilt. The ATTACH_REBUILD_LOG automatically creates a new, 1 MB log file. This file is placed in the default log-file location. For information about this location, see How to: View or Change the Default Locations for Database Files (SQL Server Management Studio).

Note

If the log files are available, the Database Engine uses those files instead of rebuilding the log files.

FOR ATTACH_REBUILD_LOG requires the following:

- A clean shutdown of the database.
- All data files (MDF and NDF) must be available.

Important

This operation breaks the log backup chain. We recommend that a full database backup be performed after the operation is completed. For more information, see BACKUP.

Typically, FOR ATTACH_REBUILD_LOG is used when you copy a read/write database with a large log to another server where the copy will be used mostly, or only, for read operations,

and therefore requires less log space than the original database.

FOR ATTACH_REBUILD_LOG cannot be specified on a database snapshot.

For more information about attaching and detaching databases, see Detaching and Attaching a Database.

<filespec>

Controls the file properties.

NAME logical_file_name

Specifies the logical name for the file. NAME is required when FILENAME is specified, except when specifying one of the FOR ATTACH clauses. A FILESTREAM filegroup cannot be named PRIMARY.

logical_file_name

Is the logical name used in SQL Server when referencing the file. Logical_file_name must be unique in the database and comply with the rules for identifiers. The name can be a character or Unicode constant, or a regular or delimited identifier.

FILENAME { 'os_file_name' | 'filestream_path' }

Specifies the operating system (physical) file name.

'os_file_name'

Is the path and file name used by the operating system when you create the file. The file must reside on one of the following devices: the local server on which SQL Server is installed, a Storage Area Network [SAN], or an iSCSI-based network. The specified path must exist before executing the CREATE DATABASE statement. For more information, see "Database Files and Filegroups" in the Remarks section.

SIZE, MAXSIZE, and FILEGROWTH parameters cannot be set when a UNC path is specified for the file.

If the file is on a raw partition, os_file_name must specify only the drive letter of an existing raw partition. Only one data file can be created on each raw partition.

Data files should not be put on compressed file systems unless the files are read-only secondary files, or the database is read-only. Log files should never be put on compressed file systems.

'filestream_path'

For a FILESTREAM filegroup, FILENAME refers to a path where FILESTREAM data will be stored. The path up to the last folder must exist, and the last folder must not exist. For example, if you specify the path C:\MyFiles\MyFilestreamData, C:\MyFiles must exist before you run ALTER DATABASE, but the MyFilestreamData folder must not exist.

The filegroup and file (`<filespec>`) must be created in the same statement.

The SIZE and FILEGROWTH properties do not apply to a FILESTREAM filegroup.

SIZE size

Specifies the size of the file.

SIZE cannot be specified when the os_file_name is specified as a UNC path. SIZE does not apply to a FILESTREAM filegroup.

size

Is the initial size of the file.

When size is not supplied for the primary file, the Database Engine uses the size of the primary file in the model database. When a secondary data file or log file is specified, but size is not specified for the file, the Database Engine makes the file 1 MB. The size specified for the primary file must be at least as large as the primary file of the model database.

The kilobyte (KB), megabyte (MB), gigabyte (GB), or terabyte (TB) suffixes can be used. The default is MB. Specify a whole number; do not include a decimal. Size is an integer value. For values greater than 2147483647, use larger units.

MAXSIZE max_size

Specifies the maximum size to which the file can grow. MAXSIZE cannot be specified when the os_file_name is specified as a UNC path.

max_size

Is the maximum file size. The KB, MB, GB, and TB suffixes can be used. The default is MB. Specify a whole number; do not include a decimal. If max_size is not specified, the file grows until the disk is full. Max_size is an integer value. For values greater than 2147483647, use larger units.

UNLIMITED

Specifies that the file grows until the disk is full. In SQL Server, a log file specified with unlimited growth has a maximum size of 2 TB, and a data file has a maximum size of 16 TB.

Note

There is no maximum size when this option is specified for a FILESTREAM container. It continues to grow until the disk is full.

FILEGROWTH growth_increment

Specifies the automatic growth increment of the file. The FILEGROWTH setting for a file cannot exceed the MAXSIZE setting. FILEGROWTH cannot be specified when the os_file_name is specified as a UNC path. FILEGROWTH does not apply to a FILESTREAM filegroup.

growth_increment

Is the amount of space added to the file every time new space is required.

The value can be specified in MB, KB, GB, TB, or percent (%). If a number is specified without an MB, KB, or % suffix, the default is MB. When % is specified, the growth increment size is the specified percentage of the size of the file at the time the increment occurs. The size specified is rounded to the nearest 64 KB.

A value of 0 indicates that automatic growth is off and no additional space is allowed.

If FILEGROWTH is not specified, the default value is 1 MB for data files and 10% for log files, and the minimum value is 64 KB.

Note

In SQL Server, the default growth increment for data files has changed from 10% to 1 MB. The log file default of 10% remains unchanged.

<filegroup>

Controls the filegroup properties. Filegroup cannot be specified on a database snapshot.

FILEGROUP filegroup_name

Is the logical name of the filegroup.

filegroup_name

filegroup_name must be unique in the database and cannot be the system-provided names PRIMARY and PRIMARY_LOG. The name can be a character or Unicode constant, or a regular or delimited identifier. The name must comply with the rules for identifiers.

CONTAINS FILESTREAM

Specifies that the filegroup stores FILESTREAM binary large objects (BLOBs) in the file system.

DEFAULT

Specifies the named filegroup is the default filegroup in the database.

database_snapshot_name

Is the name of the new database snapshot. Database snapshot names must be unique within an instance of SQL Server and comply with the rules for identifiers. database_snapshot_name can be a maximum of 128 characters.

ON (NAME = logical_file_name, FILENAME = 'os_file_name') [,... n]

For creating a database snapshot, specifies a list of files in the source database. For the snapshot to work, all the data files must be specified individually. However, log files are not allowed for database snapshots. FILESTREAM filegroups are not supported by database snapshots. If a FILESTREAM data file is included in a CREATE DATABASE ON clause, the statement will fail and an error will be raised.

For descriptions of NAME and FILENAME and their values see the descriptions of the equivalent <filespec> values.

Note

When you create a database snapshot, the other <filespec> options and the keyword PRIMARY are disallowed.

AS SNAPSHOT OF source_database_name

Specifies that the database being created is a database snapshot of the source database specified by source_database_name. The snapshot and source database must be on the same

instance.

For more information, see "Database Snapshots" in the Remarks section.

Remarks

The master database should be backed up whenever a user database is created, modified, or dropped.

The CREATE DATABASE statement must run in autocommit mode (the default transaction management mode) and is not allowed in an explicit or implicit transaction.

You can use one CREATE DATABASE statement to create a database and the files that store the database. SQL Server implements the CREATE DATABASE statement by using the following steps:

1. The SQL Server uses a copy of the model database to initialize the database and its metadata.
2. A service broker GUID is assigned to the database.
3. The Database Engine then fills the rest of the database with empty pages, except for pages that have internal data that records how the space is used in the database.

A maximum of 32,767 databases can be specified on an instance of SQL Server.

Each database has an owner that can perform special activities in the database. The owner is the user that creates the database. The database owner can be changed by using sp_changedbowner.

Database Files and Filegroups

Every database has at least two files, a *primary file* and a *transaction log file*, and at least one filegroup. A maximum of 32,767 files and 32,767 filegroups can be specified for each database.

When you create a database, make the data files as large as possible based on the maximum amount of data you expect in the database

We recommend that you use a Storage Area Network (SAN), iSCSI-based network, or locally attached disk for the storage of your SQL Server database files, because this configuration optimizes SQL Server performance and reliability.

Database Snapshots

You can use the CREATE DATABASE statement to create a read-only, static view, a *database snapshot* of the *source database*. A database snapshot is transactionally consistent with the source database as it existed at the time when the snapshot was created. A source database can have multiple snapshots.

Note

When you create a database snapshot, the CREATE DATABASE statement cannot reference log files, offline files, restoring files, and defunct files.

If creating a database snapshot fails, the snapshot becomes suspect and must be deleted. For more information, see DROP DATABASE.

Each snapshot persists until it is deleted by using DROP DATABASE.

For more information, see Database Snapshots.

Database Options

Several database options are automatically set whenever you create a database. For a list of these options, see ALTER DATABASE SET Options (Transact-SQL).

The model Database and Creating New Databases

All user-defined objects in the model database are copied to all newly created databases. You can add any objects, such as tables, views, stored procedures, data types, and so on, to the model database to be included in all newly created databases.

When a CREATE DATABASE database_name statement is specified without additional size parameters, the primary data file is made the same size as the primary file in the model database.

Unless FOR ATTACH is specified, each new database inherits the database option settings from the model database. For example, the database option auto shrink is set to **true** in model and in any new databases you create. If you change the options in the model database, these new option settings are used in any new databases you create. Changing operations in the model database does not affect existing databases. If FOR ATTACH is specified on the CREATE DATABASE statement, the new database inherits the database option settings of the original database.

Viewing Database Information

You can use catalog views, system functions, and system stored procedures to return information about databases, files, and filegroups. For more information, see System Views (Transact-SQL).

Permissions

Requires CREATE DATABASE, CREATE ANY DATABASE, or ALTER ANY DATABASE permission.

To maintain control over disk use on an instance of SQL Server, permission to create databases is typically limited to a few login accounts.

Permissions on Data and Log Files

In SQL Server, certain permissions are set on the data and log files of each database. The following permissions are set whenever the following operations are applied to a database:

Created	Modified to add a new file

Attached	Backed up
Detached	Restored

The permissions prevent the files from being accidentally tampered with if they reside in a directory that has open permissions.

Note
Microsoft SQL Server 2005 Express Edition does not set data and log file permissions.

Examples

A. Creating a database without specifying files

The following example creates the database `mytest` and creates a corresponding primary and transaction log file. Because the statement has no <filespec> items, the primary database file is the size of the model database primary file. The transaction log is set to the larger of these values: 512KB or 25% the size of the primary data file. Because MAXSIZE is not specified, the files can grow to fill all available disk space. This example also demonstrates how to drop the database named `mytest` if it exists, before creating the `mytest` database.

```
USE master;
GO
CREATE DATABASE mytest;
GO
-- Verify the database files and sizes
SELECT name, size, size*1.0/128 AS [Size in MBs]
FROM sys.master_files
WHERE name = N'mytest';
GO
```

B. Creating a database that specifies the data and transaction log files

The following example creates the database `Sales`. Because the keyword PRIMARY is not used, the first file (`Sales_dat`) becomes the primary file. Because neither MB nor KB is specified in the SIZE parameter for the `Sales_dat` file, it uses MB and is allocated in megabytes. The `Sales_log` file is allocated in megabytes because the `MB` suffix is explicitly stated in the `SIZE` parameter.

```
USE master;
GO
CREATE DATABASE Sales
ON
( NAME = Sales_dat,
```

```
    FILENAME = 'C:\Program Files\Microsoft SQL
Server\MSSQL11.MSSQLSERVER\MSSQL\DATA\saledat.mdf',
    SIZE = 10,
    MAXSIZE = 50,
    FILEGROWTH = 5 )
LOG ON
( NAME = Sales_log,
    FILENAME = 'C:\Program Files\Microsoft SQL
Server\MSSQL11.MSSQLSERVER\MSSQL\DATA\salelog.ldf',
    SIZE = 5MB,
    MAXSIZE = 25MB,
    FILEGROWTH = 5MB ) ;
GO
```

C. Creating a database by specifying multiple data and transaction log files

The following example creates the database `Archive` that has three `100-MB` data files and two `100-MB` transaction log files. The primary file is the first file in the list and is explicitly specified with the `PRIMARY` keyword. The transaction log files are specified following the `LOG ON` keywords. Note the extensions used for the files in the `FILENAME` option: `.mdf` is used for primary data files, `.ndf` is used for the secondary data files, and `.ldf` is used for transaction log files. This example places the database on the `D:` drive instead of with the `master` database.

```
USE master;
GO
CREATE DATABASE Archive
ON
PRIMARY
    (NAME = Arch1,
    FILENAME = 'D:\SalesData\archdat1.mdf',
    SIZE = 100MB,
    MAXSIZE = 200,
    FILEGROWTH = 20),
    ( NAME = Arch2,
    FILENAME = 'D:\SalesData\archdat2.ndf',
    SIZE = 100MB,
    MAXSIZE = 200,
    FILEGROWTH = 20),
```

```
    ( NAME = Arch3,
    FILENAME = 'D:\SalesData\archdat3.ndf',
    SIZE = 100MB,
    MAXSIZE = 200,
    FILEGROWTH = 20)
LOG ON
   (NAME = Archlog1,
    FILENAME = 'D:\SalesData\archlog1.ldf',
    SIZE = 100MB,
    MAXSIZE = 200,
    FILEGROWTH = 20),
   (NAME = Archlog2,
    FILENAME = 'D:\SalesData\archlog2.ldf',
    SIZE = 100MB,
    MAXSIZE = 200,
    FILEGROWTH = 20) ;
GO
```

D. Creating a database that has filegroups

The following example creates the database `Sales` that has the following filegroups:

- The primary filegroup with the files `Spri1_dat` and `Spri2_dat`. The FILEGROWTH increments for these files are specified as `15%`.
- A filegroup named `SalesGroup1` with the files `SGrp1Fi1` and `SGrp1Fi2`.
- A filegroup named `SalesGroup2` with the files `SGrp2Fi1` and `SGrp2Fi2`.

This example places the data and log files on different disks to improve performance.

```
USE master;
GO
CREATE DATABASE Sales
ON PRIMARY
( NAME = SPri1_dat,
    FILENAME = 'D:\SalesData\SPri1dat.mdf',
    SIZE = 10,
    MAXSIZE = 50,
    FILEGROWTH = 15% ),
( NAME = SPri2_dat,
```

```
    FILENAME = 'D:\SalesData\SPri2dt.ndf',
    SIZE = 10,
    MAXSIZE = 50,
    FILEGROWTH = 15% ),
FILEGROUP SalesGroup1
( NAME = SGrp1Fi1_dat,
    FILENAME = 'D:\SalesData\SG1Fi1dt.ndf',
    SIZE = 10,
    MAXSIZE = 50,
    FILEGROWTH = 5 ),
( NAME = SGrp1Fi2_dat,
    FILENAME = 'D:\SalesData\SG1Fi2dt.ndf',
    SIZE = 10,
    MAXSIZE = 50,
    FILEGROWTH = 5 ),
FILEGROUP SalesGroup2
( NAME = SGrp2Fi1_dat,
    FILENAME = 'D:\SalesData\SG2Fi1dt.ndf',
    SIZE = 10,
    MAXSIZE = 50,
    FILEGROWTH = 5 ),
( NAME = SGrp2Fi2_dat,
    FILENAME = 'D:\SalesData\SG2Fi2dt.ndf',
    SIZE = 10,
    MAXSIZE = 50,
    FILEGROWTH = 5 )
LOG ON
( NAME = Sales_log,
    FILENAME = 'E:\SalesLog\salelog.ldf',
    SIZE = 5MB,
    MAXSIZE = 25MB,
    FILEGROWTH = 5MB ) ;
GO
```

E. Attaching a database

The following example detaches the database `Archive` created in example D, and then attaches it by using the `FOR ATTACH` clause. `Archive` was defined to have multiple data and log files. However, because the location of the files has not changed since they were created, only the primary file has to be specified in the `FOR ATTACH` clause. Beginning with SQL Server 2005, any full-text files that are part of the database that is being attached will be attached with the database.

```
USE master;
GO
sp_detach_db Archive;
GO
CREATE DATABASE Archive
      ON (FILENAME = 'D:\SalesData\archdat1.mdf')
      FOR ATTACH ;
GO
```

F. Creating a database snapshot

The following example creates the database snapshot `sales_snapshot0600`. Because a database snapshot is read-only, a log file cannot be specified. In conformance with the syntax, every file in the source database is specified, and filegroups are not specified.

The source database for this example is the `Sales` database created in example D.

```
USE master;
GO
CREATE DATABASE sales_snapshot0600 ON
    ( NAME = SPri1_dat, FILENAME = 'D:\SalesData\SPri1dat_0600.ss'),
    ( NAME = SPri2_dat, FILENAME = 'D:\SalesData\SPri2dt_0600.ss'),
    ( NAME = SGrp1Fi1_dat, FILENAME = 'D:\SalesData\SG1Fi1dt_0600.ss'),
    ( NAME = SGrp1Fi2_dat, FILENAME = 'D:\SalesData\SG1Fi2dt_0600.ss'),
    ( NAME = SGrp2Fi1_dat, FILENAME = 'D:\SalesData\SG2Fi1dt_0600.ss'),
    ( NAME = SGrp2Fi2_dat, FILENAME = 'D:\SalesData\SG2Fi2dt_0600.ss')
AS SNAPSHOT OF Sales ;
GO
```

G. Creating a database and specifying a collation name and options

The following example creates the database `MyOptionsTest`. A collation name is specified and the `TRUSTYWORTHY` and `DB_CHAINING` options are set to `ON`.

```
USE master;
```

```
GO
IF DB_ID (N'MyOptionsTest') IS NOT NULL
DROP DATABASE MyOptionsTest;
GO
CREATE DATABASE MyOptionsTest
COLLATE French_CI_AI
WITH TRUSTWORTHY ON, DB_CHAINING ON;
GO
--Verifying collation and option settings.
SELECT name, collation_name, is_trustworthy_on, is_db_chaining_on
FROM sys.databases
WHERE name = N'MyOptionsTest';
GO
```

H. Attaching a full-text catalog that has been moved

The following example attaches the full-text catalog `AdvWksFtCat` along with the `AdventureWorks2012` data and log files. In this example, the full-text catalog is moved from its default location to a new location `c:\myFTCatalogs`. The data and log files remain in their default locations.

```
USE master;
GO
--Detach the AdventureWorks2012 database
sp_detach_db AdventureWorks2012;
GO
-- Physically move the full text catalog to the new location.
--Attach the AdventureWorks2012 database and specify the new location of the
full-text catalog.
CREATE DATABASE AdventureWorks2012 ON
    (FILENAME = 'c:\Program Files\Microsoft SQL
Server\MSSQL11.MSSQLSERVER\MSSQL\Data\AdventureWorks2012_data.mdf'),
    (FILENAME = 'c:\Program Files\Microsoft SQL
Server\MSSQL11.MSSQLSERVER\MSSQL\Data\AdventureWorks2012_log.ldf'),
    (FILENAME = 'c:\myFTCatalogs\AdvWksFtCat')
FOR ATTACH;
GO
```

I. Creating a database that specifies a row filegroup and two FILESTREAM filegroups

The following example creates the `FileStreamDB` database. The database is created with one row filegroup and two FILESTREAM filegroups. Each filegroup contains one file:

- `FileStreamDB_data` contains row data. It contains one file, `FileStreamDB_data.mdf` with the default path.
- `FileStreamPhotos` contains FILESTREAM data. It contains two FILESTREAM data containers, `FSPhotos`, located at `C:\MyFSfolder\Photos` and `FSPhotos2`, located at `D:\MyFSfolder\Photos`. It is marked as the default FILESTREAM filegroup.
- `FileStreamResumes` contains FILESTREAM data. It contains one FILESTREAM data container, `FSResumes`, located at `C:\MyFSfolder\Resumes`.

```
USE master;
GO
-- Get the SQL Server data path.
DECLARE @data_path nvarchar(256);
SET @data_path = (SELECT SUBSTRING(physical_name, 1, CHARINDEX(N'master.mdf',
LOWER(physical_name)) - 1)
                  FROM master.sys.master_files
                  WHERE database_id = 1 AND file_id = 1);

 -- Execute the CREATE DATABASE statement.
EXECUTE ('CREATE DATABASE FileStreamDB
ON PRIMARY
    (
    NAME = FileStreamDB_data
    ,FILENAME = ''' + @data_path + 'FileStreamDB_data.mdf''
    ,SIZE = 10MB
    ,MAXSIZE = 50MB
    ,FILEGROWTH = 15%
    ),
FILEGROUP FileStreamPhotos CONTAINS FILESTREAM DEFAULT
    (
    NAME = FSPhotos
    ,FILENAME = ''C:\MyFSfolder\Photos''
-- SIZE and FILEGROWTH should not be specified here.
```

```
-- If they are specified an error will be raised.
, MAXSIZE = 5000 MB
    ),
    (
      NAME = FSPhotos2
      , FILENAME = ''D:\MyFSfolder\Photos''
      , MAXSIZE = 10000 MB
     ),
FILEGROUP FileStreamResumes CONTAINS FILESTREAM
    (
    NAME = FileStreamResumes
    ,FILENAME = ''C:\MyFSfolder\Resumes''
    )
LOG ON
    (
    NAME = FileStream_log
    ,FILENAME = ''' + @data_path + 'FileStreamDB_log.ldf''
    ,SIZE = 5MB
    ,MAXSIZE = 25MB
    ,FILEGROWTH = 5MB
    )'
);
GO
```

J. Creating a database that has a FILESTREAM filegroup with multiple files

The following example creates the `BlobStore1` database. The database is created with one row filegroup and one FILESTREAM filegroup, `FS`. The FILESTREAM filegroup contains two files, `FS1` and `FS2`. Then the database is altered by adding a third file, `FS3`, to the FILESTREAM filegroup.

```
USE [master]
GO

CREATE DATABASE [BlobStore1]
CONTAINMENT = NONE
ON PRIMARY
(
```

```
    NAME = N'BlobStore1',
    FILENAME = N'C:\BlobStore\BlobStore1.mdf',
    SIZE = 100MB,
    MAXSIZE = UNLIMITED,
    FILEGROWTH = 1MB
),
FILEGROUP [FS] CONTAINS FILESTREAM DEFAULT
(
    NAME = N'FS1',
    FILENAME = N'C:\BlobStore\FS1',
    MAXSIZE = UNLIMITED
),
(
    NAME = N'FS2',
    FILENAME = N'C:\BlobStore\FS2',
    MAXSIZE = 100MB
)
LOG ON
(
    NAME = N'BlobStore1_log',
    FILENAME = N'C:\BlobStore\BlobStore1_log.ldf',
    SIZE = 100MB,
    MAXSIZE = 1GB,
    FILEGROWTH = 1MB
)
GO

ALTER DATABASE [BlobStore1]
ADD FILE
(
    NAME = N'FS3',
    FILENAME = N'C:\BlobStore\FS3',
    MAXSIZE = 100MB
```

```
)
TO FILEGROUP [FS]
GO
```

See Also

ALTER DATABASE (Transact-SQL)
Detaching and Attaching a Database
DROP DATABASE
eventdata (Transact-SQL)
sp_changedbowner (Transact-SQL)
sp_detach_db (Transact-SQL)
sp_removedbreplication (Transact-SQL)
Database Snapshots
Moving Database Files
Databases
Designing and Implementing FILESTREAM Storage

CREATE DATABASE AUDIT SPECIFICATION

Creates a database audit specification object using the SQL Server audit feature. For more information, see Understanding SQL Server Audit.

 Transact-SQL Syntax Conventions

Syntax

```
CREATE DATABASE AUDIT SPECIFICATION audit_specification_name
{
   FOR SERVER AUDIT audit_name
     [ { ADD ( { <audit_action_specification> | audit_action_group_name } )
    } [ , ...n] ]
   [ WITH ( STATE = { ON | OFF } ) ]
}
[ ; ]
<audit_action_specification>::=
{
    action [ ,...n ]ON [ class :: ] securable BY principal [ ,...n ]
}
```

Arguments

audit_specification_name

Is the name of the audit specification.

audit_name

Is the name of the audit to which this specification is applied.

audit_action_specification

Is the specification of actions on securables by principals that should be recorded in the audit.

action

Is the name of one or more database-level auditable actions. For a list of audit actions, see SQL Server Audit Action Groups and Actions.

audit_action_group_name

Is the name of one or more groups of database-level auditable actions. For a list of audit action groups, see SQL Server Audit Action Groups and Actions.

class

Is the class name (if applicable) on the securable.

securable

Is the table, view, or other securable object in the database on which to apply the audit action or audit action group. For more information, see Securables.

principal

Is the name of SQL Server principal on which to apply the audit action or audit action group. For more information, see Principals (Database Engine).

WITH (STATE = { ON | OFF })

Enables or disables the audit from collecting records for this audit specification.

Remarks

Database audit specifications are non-securable objects that reside in a given database. When a database audit specification is created, it is in a disabled state.

Permissions

Users with the ALTER ANY DATABASE AUDIT permission can create database audit specifications and bind them to any audit.

After a database audit specification is created, it can be viewed by principals with the CONTROL SERVER, ALTER ANY DATABASE AUDIT permissions, or the sysadmin account.

Examples

The following example creates a server audit called `Payrole_Security_Audit` and then a database audit specification called `Payrole_Security_Audit` that audits `SELECT` and `INSERT` statements by the `dbo` user, for the `HumanResources.EmployeePayHistory` table in the `AdventureWorks2012` database.

```
USE master ;
GO
-- Create the server audit.
CREATE SERVER AUDIT Payrole_Security_Audit
    TO FILE ( FILEPATH =
'C:\Program Files\Microsoft SQL Server\MSSQL10_50.MSSQLSERVER\MSSQL\DATA' ) ;
GO
-- Enable the server audit.
ALTER SERVER AUDIT Payrole_Security_Audit
WITH (STATE = ON) ;
GO
-- Move to the target database.
USE AdventureWorks2012 ;
GO
-- Create the database audit specification.
CREATE DATABASE AUDIT SPECIFICATION Audit_Pay_Tables
FOR SERVER AUDIT Payrole_Security_Audit
ADD (SELECT , INSERT
     ON HumanResources.EmployeePayHistory BY dbo )
WITH (STATE = ON) ;
GO
```

See Also

CREATE SERVER AUDIT (Transact-SQL)
ALTER SERVER AUDIT (Transact-SQL)
DROP SERVER AUDIT (Transact-SQL)
CREATE SERVER AUDIT SPECIFICATION (Transact-SQL)
ALTER SERVER AUDIT SPECIFICATION (Transact-SQL)
DROP SERVER AUDIT SPECIFICATION (Transact-SQL)
CREATE DATABASE AUDIT SPECIFICATION (Transact-SQL)
ALTER DATABASE AUDIT SPECIFICATION (Transact-SQL)

DROP DATABASE AUDIT SPECIFICATION (Transact-SQL)
ALTER AUTHORIZATION (Transact-SQL)
fn_get_audit_file (Transact-SQL)
sys.server_audits (Transact-SQL)
sys.server_file_audits (Transact-SQL)
sys.server_audit_specifications (Transact-SQL)
sys.server_audit_specifications_details (Transact-SQL)
sys.database_audit_specifications (Transact-SQL)
sys.audit_database_specification_details (Transact-SQL)
sys.dm_server_audit_status
sys.dm_audit_actions
Create a Server Audit and Server Audit Specification

CREATE DATABASE ENCRYPTION KEY

Creates an encryption key that is used for transparently encrypting a database. For more information about transparent database encryption, see Understanding Transparent Data Encryption (TDE).

 Transact-SQL Syntax Conventions

Syntax

```
CREATE DATABASE ENCRYPTION KEY
    WITH ALGORITHM = { AES_128 | AES_192 | AES_256 | TRIPLE_DES_3KEY }
  ENCRYPTION BY SERVER
  {
    CERTIFICATE Encryptor_Name |
    ASYMMETRIC KEY Encryptor_Name
  }
[ ; ]
```

Arguments

WITH ALGORITHM = { AES_128 | AES_192 | AES_256 | TRIPLE_DES_3KEY }

Specifies the encryption algorithm that is used for the encryption key.

ENCRYPTION BY SERVER CERTIFICATE Encryptor_Name

Specifies the name of the encryptor used to encrypt the database encryption key.

ENCRYPTION BY SERVER ASYMMETRIC KEY Encryptor_Name

Specifies the name of the asymmetric key used to encrypt the database encryption key. In order to encrypt the database encryption key with an asymmetric key, the asymmetric key must reside on an extensible key management provider.

Remarks

A database encryption key is required before a database can be encrypted by using *Transparent Database Encryption* (TDE). When a database is transparently encrypted, the whole database is encrypted at the file level, without any special code modifications. The certificate or asymmetric key that is used to encrypt the database encryption key must be located in the master system database.

Database encryption statements are allowed only on user databases.

The database encryption key cannot be exported from the database. It is available only to the system, to users who have debugging permissions on the server, and to users who have access to the certificates that encrypt and decrypt the database encryption key.

The database encryption key does not have to be regenerated when a database owner (dbo) is changed.

Permissions

Requires CONTROL permission on the database and VIEW DEFINITION permission on the certificate or asymmetric key that is used to encrypt the database encryption key.

Examples

For additional examples using TDE, see Understanding Transparent Data Encryption (TDE) and How to: Enable TDE using EKM.

The following example creates a database encryption key by using the `AES_256` algorithm, and protects the private key with a certificate named `MyServerCert`.

```
USE AdventureWorks2012;
GO
CREATE DATABASE ENCRYPTION KEY
WITH ALGORITHM = AES_256
ENCRYPTION BY SERVER CERTIFICATE MyServerCert;
GO
```

See Also

Understanding Transparent Data Encryption (TDE)

SQL Server Encryption

SQL Server and Database Encryption Keys (Database Engine)

Encryption Hierarchy

ALTER DATABASE SET Options (Transact-SQL)

ALTER DATABASE ENCRYPTION KEY (Transact-SQL)
DROP DATABASE ENCRYPTION KEY (Transact-SQL)
sys.dm_database_encryption_keys

CREATE DEFAULT

Creates an object called a default. When bound to a column or an alias data type, a default specifies a value to be inserted into the column to which the object is bound (or into all columns, in the case of an alias data type), when no value is explicitly supplied during an insert.

Important
This feature will be removed in a future version of Microsoft SQL Server. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use default definitions created using the DEFAULT keyword of ALTER TABLE or CREATE TABLE.

Transact-SQL Syntax Conventions

Syntax

```
CREATE DEFAULT [ schema_name . ] default_name
AS constant_expression [ ; ]
```

Arguments

schema_name

Is the name of the schema to which the default belongs.

default_name

Is the name of the default. Default names must conform to the rules for identifiers. Specifying the default owner name is optional.

constant_expression

Is an expression that contains only constant values (it cannot include the names of any columns or other database objects). Any constant, built-in function, or mathematical expression can be used, except those that contain alias data types. User-defined functions cannot be used.. Enclose character and date constants in single quotation marks ('); monetary, integer, and floating-point constants do not require quotation marks. Binary data must be preceded by 0x, and monetary data must be preceded by a dollar sign ($). The default value must be compatible with the column data type.

Remarks

A default name can be created only in the current database. Within a database, default names must be unique by schema. When a default is created, use **sp_bindefault** to bind it to a column or to an alias data type.

If the default is not compatible with the column to which it is bound, SQL Server generates an error message when trying to insert the default value. For example, N/A cannot be used as a default for a **numeric** column.

If the default value is too long for the column to which it is bound, the value is truncated.

CREATE DEFAULT statements cannot be combined with other Transact-SQL statements in a single batch.

A default must be dropped before creating a new one of the same name, and the default must be unbound by executing **sp_unbindefault** before it is dropped.

If a column has both a default and a rule associated with it, the default value must not violate the rule. A default that conflicts with a rule is never inserted, and SQL Server generates an error message each time it attempts to insert the default.

When bound to a column, a default value is inserted when:

- A value is not explicitly inserted.
- Either the DEFAULT VALUES or DEFAULT keywords are used with INSERT to insert default values.

If NOT NULL is specified when creating a column and a default is not created for it, an error message is generated when a user fails to make an entry in that column. The following table illustrates the relationship between the existence of a default and the definition of a column as NULL or NOT NULL. The entries in the table show the result.

Column definition	No entry, no default	No entry, default	Enter NULL, no default	Enter NULL, default
NULL	NULL	default	NULL	NULL
NOT NULL	Error	default	error	error

To rename a default, use **sp_rename**. For a report on a default, use **sp_help**.

Permissions

To execute CREATE DEFAULT, at a minimum, a user must have CREATE DEFAULT permission in the current database and ALTER permission on the schema in which the default is being created.

Examples

A. Creating a simple character default

The following example creates a character default called `unknown`.

```
USE AdventureWorks2012;
GO
CREATE DEFAULT phonedflt AS 'unknown';
```

B. Binding a default

The following example binds the default created in example A. The default takes effect only if no entry is specified for the `Phone` column of the `Contact` table. Note that omitting any entry is different from explicitly stating NULL in an INSERT statement.

Because a default named `phonedflt` does not exist, the following Transact-SQL statement fails. This example is for illustration only.

```
USE AdventureWorks2012;
GO
sp_bindefault 'phonedflt', 'Person.PersonPhone.PhoneNumber';
```

See Also

ALTER TABLE
CREATE RULE
CREATE TABLE
DROP DEFAULT
DROP RULE
Expressions
INSERT
sp_bindefault
sp_help
sp_helptext
sp_rename
sp_unbindefault

CREATE ENDPOINT

Creates endpoints and defines their properties, including the methods available to client applications. For related permissions information, see Sample XML Applications.

The syntax for CREATE ENDPOINT can logically be broken into two parts:

- The first part starts with AS and ends before the FOR clause.

  In this part, you provide information specific to the transport protocol (TCP) and set a listening port number for the endpoint, as well as the method of endpoint authentication and/or a list of IP addresses (if any) that you want to restrict from accessing the endpoint.
- The second part starts with the FOR clause.

  In this part, you define the payload that is supported on the endpoint. The payload can be one of several supported types: Transact-SQL, service broker, database mirroring. In this part, you also include language-specific information.

Note
Native XML Web Services (SOAP/HTTP endpoints) was removed in SQL Server 2012.

Transact-SQL Syntax Conventions

Syntax

```
CREATE ENDPOINT endPointName [ AUTHORIZATION login ]
[ STATE = { STARTED | STOPPED | DISABLED } ]
AS { TCP } (
   <protocol_specific_arguments>
      )
FOR { TSQL | SERVICE_BROKER | DATABASE_MIRRORING } (
   <language_specific_arguments>
      )

<AS TCP_protocol_specific_arguments> ::=
AS TCP (
  LISTENER_PORT = listenerPort
  [ [ , ] LISTENER_IP = ALL | ( 4-part-ip ) | ( "ip_address_v6" ) ]

)

<FOR SERVICE_BROKER_language_specific_arguments> ::=
FOR SERVICE_BROKER (
   [ AUTHENTICATION = {
         WINDOWS [ { NTLM | KERBEROS | NEGOTIATE } ]
      | CERTIFICATE certificate_name
      | WINDOWS [ { NTLM | KERBEROS | NEGOTIATE } ] CERTIFICATE certificate_name
      | CERTIFICATE certificate_name WINDOWS [ { NTLM | KERBEROS | NEGOTIATE } ]
    } ]
   [ [ , ] ENCRYPTION = { DISABLED | { { SUPPORTED | REQUIRED }
      [ ALGORITHM { RC4 | AES | AES RC4 | RC4 AES } ] }
   ]
   [ [ , ] MESSAGE_FORWARDING = { ENABLED | DISABLED } ]
   [ [ , ] MESSAGE_FORWARD_SIZE = forward_size ]
)
```

<FOR DATABASE_MIRRORING_language_specific_arguments> ::=

```
FOR DATABASE_MIRRORING (
  [ AUTHENTICATION = {
        WINDOWS [ { NTLM | KERBEROS | NEGOTIATE } ]
    | CERTIFICATE certificate_name
    | WINDOWS [ { NTLM | KERBEROS | NEGOTIATE } ] CERTIFICATE certificate_name
    | CERTIFICATE certificate_name WINDOWS [ { NTLM | KERBEROS | NEGOTIATE } ]
  [ [ [ , ] ] ENCRYPTION = { DISABLED | { { SUPPORTED | REQUIRED }
     [ ALGORITHM { RC4 | AES | AES RC4 | RC4 AES } ] }

   ]
  [ , ] ROLE = { WITNESS | PARTNER | ALL }
)
```

Arguments

endPointName

Is the assigned name for the endpoint you are creating. Use when updating or deleting the endpoint.

AUTHORIZATION login

Specifies a valid SQL Server or Windows login that is assigned ownership of the newly created endpoint object. If AUTHORIZATION is not specified, by default, the caller becomes owner of the newly created object.

To assign ownership by specifying AUTHORIZATION, the caller must have IMPERSONATE permission on the specified login.

To reassign ownership, see ALTER ENDPOINT (Transact-SQL).

STATE = { STARTED | STOPPED | DISABLED }

Is the state of the endpoint when it is created. If the state is not specified when the endpoint is created, STOPPED is the default.

STARTED

Endpoint is started and is actively listening for connections.

DISABLED

Endpoint is disabled. In this state, the server listens to port requests but returns errors to

clients.

STOPPED

Endpoint is stopped. In this state, the server does not listen to the endpoint port or respond to any attempted requests to use the endpoint.

To change the state, use ALTER ENDPOINT.

AS { TCP }

Specifies the transport protocol to use.

FOR { TSQL | SERVICE_BROKER | DATABASE_MIRRORING }

Specifies the payload type.

Currently, there are no Transact-SQL language-specific arguments to pass in the `<language_specific_arguments>` parameter.

TCP Protocol Option

The following arguments apply only to the TCP protocol option.

LISTENER_PORT = listenerPort

Specifies the port number listened to for connections by the service broker TCP/IP protocol. By convention, 4022 is used but any number between 1024 and 32767 is valid.

LISTENER_IP = ALL | (4-part-ip) | ("ip_address_v6")

Specifies the IP address that the endpoint will listen on. The default is ALL. This means that the listener will accept a connection on any valid IP address.

If you configure database mirroring with an IP address instead of a fully-qualified domain name (`ALTER DATABASE SET PARTNER = partner_IP_address` or `ALTER DATABASE SET WITNESS = witness_IP_address`), you have to specify `LISTENER_IP =IP_address` instead of `LISTENER_IP=ALL` when you create mirroring endpoints.

SERVICE_BROKER and DATABASE_MIRRORING Options

The following AUTHENTICATION and ENCRYPTION arguments are common to the SERVICE_BROKER and DATABASE_MIRRORING options.

Note

For options that are specific to SERVICE_BROKER, see "SERVICE_BROKER Options," later in this section. For options that are specific to DATABASE_MIRRORING, see "DATABASE_MIRRORING Options," later in this section.

AUTHENTICATION = <authentication_options>

Specifies the TCP/IP authentication requirements for connections for this endpoint. The default is WINDOWS.

The supported authentication methods include NTLM and or Kerberos or both.

Important

All mirroring connections on a server instance use a single database mirroring endpoint. Any attempt to create an additional database mirroring endpoint will fail.

<authentication_options> ::=

WINDOWS [{ NTLM | KERBEROS | NEGOTIATE }]

Specifies that the endpoint is to connect using Windows Authentication protocol to authenticate the endpoints. This is the default.

If you specify an authorization method (NTLM or KERBEROS), that method is always used as the authentication protocol. The default value, NEGOTIATE, causes the endpoint to use the Windows negotiation protocol to choose either NTLM or Kerberos.

CERTIFICATE certificate_name

Specifies that the endpoint is to authenticate the connection using the certificate specified by certificate_name to establish identity for authorization. The far endpoint must have a certificate with the public key matching the private key of the specified certificate.

WINDOWS [{ NTLM | KERBEROS | NEGOTIATE }] CERTIFICATE certificate_name

Specifies that endpoint is to try to connect by using Windows Authentication and, if that attempt fails, to then try using the specified certificate.

CERTIFICATE certificate_name WINDOWS [{ NTLM | KERBEROS | NEGOTIATE }]

Specifies that endpoint is to try to connect by using the specified certificate and, if that attempt fails, to then try using Windows Authentication.

ENCRYPTION = { DISABLED | SUPPORTED | REQUIRED } [ALGORITHM { RC4 | AES | AES RC4 | RC4 AES }]

Specifies whether encryption is used in the process. The default is REQUIRED.

DISABLED

Specifies that data sent over a connection is not encrypted.

SUPPORTED

Specifies that the data is encrypted only if the opposite endpoint specifies either SUPPORTED or REQUIRED.

REQUIRED

Specifies that connections to this endpoint must use encryption. Therefore, to connect to this endpoint, another endpoint must have ENCRYPTION set to either SUPPORTED or REQUIRED.

Optionally, you can use the ALGORITHM argument to specify the form of encryption used by the endpoint, as follows:

RC4

Specifies that the endpoint must use the RC4 algorithm. This is the default.

Note

The RC4 algorithm is only supported for backward compatibility. New material can only be encrypted using RC4 or RC4_128 when the database is in compatibility level 90 or 100. (Not recommended.) Use a newer algorithm such as one of the AES algorithms instead. In SQL Server 2012 material encrypted using RC4 or RC4_128 can be decrypted in any compatibility level.

AES

Specifies that the endpoint must use the AES algorithm.

AES RC4

Specifies that the two endpoints will negotiate for an encryption algorithm with this endpoint giving preference to the AES algorithm.

RC4 AES

Specifies that the two endpoints will negotiate for an encryption algorithm with this endpoint giving preference to the RC4 algorithm.

Note

The RC4 algorithm is deprecated. This feature will be removed in a future version of Microsoft SQL Server. Do not use this feature in new development work, and modify applications that currently use this feature as soon as possible. We recommend that you use AES.

If both endpoints specify both algorithms but in different orders, the endpoint accepting the connection wins.

SERVICE_BROKER Options

The following arguments are specific to the SERVICE_BROKER option.

MESSAGE_FORWARDING = { ENABLED | DISABLED }

Determines whether messages received by this endpoint that are for services located elsewhere will be forwarded.

ENABLED

Forwards messages if a forwarding address is available.

DISABLED

Discards messages for services located elsewhere. This is the default.

MESSAGE_FORWARD_SIZE = forward_size

Specifies the maximum amount of storage in megabytes to allocate for the endpoint to use when storing messages that are to be forwarded.

DATABASE_MIRRORING Options

The following argument is specific to the DATABASE_MIRRORING option.

ROLE = { WITNESS | PARTNER | ALL }

Specifies the database mirroring role or roles that the endpoint supports.

WITNESS

Enables the endpoint to perform in the role of a witness in the mirroring process.

Note

For SQL Server 2005 Express Edition, WITNESS is the only option available.

PARTNER

Enables the endpoint to perform in the role of a partner in the mirroring process.

ALL

Enables the endpoint to perform in the role of both a witness and a partner in the mirroring process.

For more information about these roles, see Overview of Database Mirroring.

Note

There is no default port for DATABASE_MIRRORING.

Remarks

ENDPOINT DDL statements cannot be executed inside a user transaction. ENDPOINT DDL statements do not fail even if an active snapshot isolation level transaction is using the endpoint being altered.

Requests can be executed against an ENDPOINT by the following:

- Members of **sysadmin** fixed server role
- The owner of the endpoint
- Users or groups that have been granted CONNECT permission on the endpoint

Permissions

Requires CREATE ENDPOINT permission, or membership in the **sysadmin** fixed server role. For more information, see GRANT Endpoint Permissions (Transact-SQL).

Example

Creating a database mirroring endpoint

The following example creates a database mirroring endpoint. The endpoint uses port number `7022`, although any available port number would work. The endpoint is configured to use Windows Authentication using only Kerberos. The `ENCRYPTION` option is configured to the nondefault value of `SUPPORTED` to support encrypted or unencrypted data. The endpoint is being configured to support both the partner and witness roles.

```
CREATE ENDPOINT endpoint_mirroring
    STATE = STARTED
    AS TCP ( LISTENER_PORT = 7022 )
    FOR DATABASE_MIRRORING (
       AUTHENTICATION = WINDOWS KERBEROS,
```

```
        ENCRYPTION = SUPPORTED,
        ROLE=ALL);
GO
```

See Also

ALTER ENDPOINT (Transact-SQL)
Choosing an Encryption Algorithm
DROP ENDPOINT (Transact-SQL)
EVENTDATA (Transact-SQL)

CREATE EVENT NOTIFICATION

Creates an object that sends information about a database or server event to a service broker service. Event notifications are created only by using Transact-SQL statements.

 Transact-SQL Syntax Conventions

Syntax

```
CREATE EVENT NOTIFICATION event_notification_name
ON { SERVER | DATABASE | QUEUE queue_name }
[ WITH FAN_IN ]
FOR { event_type | event_group } [ ,...n ]
TO SERVICE 'broker_service' , { 'broker_instance_specifier' | 'current database' }
[ ; ]
```

Arguments

event_notification_name

Is the name of the event notification. An event notification name must comply with the rules for identifiers and must be unique within the scope in which they are created: SERVER, DATABASE, or object_name.

SERVER

Applies the scope of the event notification to the current instance of SQL Server. If specified, the notification fires whenever the specified event in the FOR clause occurs anywhere in the instance of SQL Server.

 **Note**
This option is not available in a contained database.

DATABASE

Applies the scope of the event notification to the current database. If specified, the

notification fires whenever the specified event in the FOR clause occurs in the current database.

QUEUE

Applies the scope of the notification to a specific queue in the current database. QUEUE can be specified only if FOR QUEUE_ACTIVATION or FOR BROKER_QUEUE_DISABLED is also specified.

queue_name

Is the name of the queue to which the event notification applies. queue_name can be specified only if QUEUE is specified.

WITH FAN_IN

Instructs SQL Server to send only one message per event to any specified service for all event notifications that:

- Are created on the same event.
- Are created by the same principal (as identified by the same SID).
- Specify the same service and broker_instance_specifier.
- Specify WITH FAN_IN.

For example, three event notifications are created. All event notifications specify FOR ALTER_TABLE, WITH FAN_IN, the same TO SERVICE clause, and are created by the same SID. When an ALTER TABLE statement is run, the messages that are created by these three event notifications are merged into one. Therefore, the target service receives only one message of the event.

event_type

Is the name of an event type that causes the event notification to execute. event_type can be a Transact-SQL DDL event type, a SQL Trace event type, or a Service Broker event type. For a list of qualifying Transact-SQL DDL event types, see DDL Events. Service Broker event types are QUEUE_ACTIVATION and BROKER_QUEUE_DISABLED. For more information, see Event Notifications.

event_group

Is the name of a predefined group of Transact-SQL or SQL Trace event types. An event notification can fire after execution of any event that belongs to an event group. For a list of DDL event groups, the Transact-SQL events they cover, and the scope at which they can be defined, see DDL Event Groups.

event_group also acts as a macro, when the CREATE EVENT NOTIFICATION statement finishes, by adding the event types it covers to the **sys.events** catalog view.

'broker_service'

Specifies the target service that receives the event instance data. SQL Server opens one or more conversations to the target service for the event notification. This service must honor

the same SQL Server Events message type and contract that is used to send the message.

The conversations remain open until the event notification is dropped. Certain errors could cause the conversations to close earlier. Ending some or all conversations explicitly might prevent the target service from receiving more messages.

{ 'broker_instance_specifier' | 'current database' }

Specifies a service broker instance against which broker_service is resolved. The value for a specific service broker can be acquired by querying the **service_broker_guid** column of the **sys.databases** catalog view. Use **'current database'** to specify the service broker instance in the current database. **'current database'** is a case-insensitive string literal.

Note

This option is not available in a contained database.

Remarks

Service Broker includes a message type and contract specifically for event notifications. Therefore, a Service Broker initiating service does not have to be created because one already exists that specifies the following contract name:

`http://schemas.microsoft.com/SQL/Notifications/PostEventNotification`

The target service that receives event notifications must honor this preexisting contract.

Important

Service Broker dialog security should be configured for event notifications that send messages to a service broker on a remote server. Dialog security must be configured manually according to the full security model. For more information, see Dialog Security for Event Notifications.

If an event transaction that activates a notification is rolled back, the sending of the event notification is also rolled back. Event notifications do not fire by an action defined in a trigger when the transaction is committed or rolled back inside the trigger. Because trace events are not bound by transactions, event notifications based on trace events are sent regardless of whether the transaction that activates them is rolled back.

If the conversation between the server and the target service is broken after an event notification fires, an error is reported and the event notification is dropped.

The event transaction that originally started the notification is not affected by the success or failure of the sending of the event notification.

Any failure to send an event notification is logged.

Permissions

To create an event notification that is scoped to the database (ON DATABASE), requires CREATE DATABASE DDL EVENT NOTIFICATION permission in the current database.

To create an event notification on a DDL statement that is scoped to the server (ON SERVER), requires CREATE DDL EVENT NOTIFICATION permission in the server.

To create an event notification on a trace event, requires CREATE TRACE EVENT NOTIFICATION permission in the server.

To create an event notification that is scoped to a queue, requires ALTER permission on the queue.

Examples

Note

- In Examples A and B below, the GUID in the `TO SERVICE 'NotifyService'` clause ('8140a771-3c4b-4479-8ac0-81008ab17984') is specific to the computer on which the example was set up. For that instance, that was the GUID for the AdventureWorks2012 database.
- To copy and run these examples, you need to replace this GUID with one from your computer and SQL Server instance. As explained in the Arguments section above, you can acquire the 'broker_instance_specifier' by querying the service_broker_guid column of the sys.databases catalog view.

A. Creating an event notification that is server scoped

The following example creates the required objects to set up a target service using Service Broker. The target service references the message type and contract of the initiating service specifically for event notifications. Then an event notification is created on that target service that sends a notification whenever an `Object_Created` trace event happens on the instance of SQL Server.

```
--Create a queue to receive messages.
CREATE QUEUE NotifyQueue ;
GO
--Create a service on the queue that references
--the event notifications contract.
CREATE SERVICE NotifyService
ON QUEUE NotifyQueue
([http://schemas.microsoft.com/SQL/Notifications/PostEventNotification]);
GO
--Create a route on the service to define the address
--to which Service Broker sends messages for the service.
CREATE ROUTE NotifyRoute
WITH SERVICE_NAME = 'NotifyService',
ADDRESS = 'LOCAL';
GO
--Create the event notification.
CREATE EVENT NOTIFICATION log_ddl1
```

```
ON SERVER
FOR Object_Created
TO SERVICE 'NotifyService',
    '8140a771-3c4b-4479-8ac0-81008ab17984' ;
```

B. Creating an event notification that is database scoped

The following example creates an event notification on the same target service as the previous example. The event notification fires after an `ALTER_TABLE` event occurs on the AdventureWorks2012 sample database.

```
CREATE EVENT NOTIFICATION Notify_ALTER_T1
ON DATABASE
FOR ALTER_TABLE
TO SERVICE 'NotifyService',
    '8140a771-3c4b-4479-8ac0-81008ab17984';
```

C. Getting information about an event notification that is server scoped

The following example queries the `sys.server_event_notifications` catalog view for metadata about event notification `log_ddl1` that was created with server scope.

```
SELECT * FROM sys.server_event_notifications
WHERE name = 'log_ddl1';
```

D. Getting information about an event notification that is database scoped

The following example queries the `sys.event_notifications` catalog view for metadata about event notification `Notify_ALTER_T1` that was created with database scope.

```
SELECT * FROM sys.event_notifications
WHERE name = 'Notify_ALTER_T1';
```

See Also

Event Notifications
DROP EVENT NOTIFICATION (Transact-SQL)
EVENTDATA (Transact-SQL)
sys.event_notifications (Transact-SQL)
sys.server_event_notifications (Transact-SQL)
sys.events (Transact-SQL)
sys.server_events (Transact-SQL)

CREATE EVENT SESSION

Creates an Extended Events session that identifies the source of the events, the event session targets, and the event session options.

Transact-SQL Syntax Conventions.

Syntax

```
CREATE EVENT SESSION event_session_name
ON SERVER
{
   <event_definition> [ ,...n]
   [ <event_target_definition> [ ,...n] ]
   [ WITH ( <event_session_options> [ ,...n] ) ]
}
;

<event_definition>::=
{
   ADD EVENT [event_module_guid].event_package_name.event_name
      [ ( {
            [ SET { event_customizable_attribute = <value> [ ,...n] } ]
            [ ACTION ( { [event_module_guid].event_package_name.action_name [ ,...n] } ) ]
            [ WHERE <predicate_expression> ]
      } ) ]
}

<predicate_expression> ::=
{
   [ NOT ] <predicate_factor> | { ( <predicate_expression> ) }
   [ { AND | OR } [ NOT ] { <predicate_factor> | ( <predicate_expression> ) } ]
   [ ,...n ]
}

<predicate_factor>::=
{
   <predicate_leaf> | ( <predicate_expression> )
}

<predicate_leaf>::=
```

{

<predicate_source_declaration> { = | < > | ! = | > | > = | < | < = } <value>

| [**event_module_guid**].**event_package_name**.**predicate_compare_name** **(** <predicate_source_declaration>, <value> **)**

}

<predicate_source_declaration>::=

{

event_field_name | **(** [**event_module_guid**].**event_package_name**.**predicate_source_name** **)**

}

<value>::=

{

number | **'string'**

}

<event_target_definition>::=

{

ADD TARGET [**event_module_guid**].**event_package_name**.**target_name**

[**(** SET { **target_parameter_name** = <value> [**,**...**n**] } **)**]

}

<event_session_options>::=

{

[MAX_MEMORY = **size** [KB | MB]]

[[,] EVENT_RETENTION_MODE = { ALLOW_SINGLE_EVENT_LOSS | ALLOW_MULTIPLE_EVENT_LOSS | NO_EVENT_LOSS }]

[[,] MAX_DISPATCH_LATENCY = { **seconds** SECONDS | INFINITE }]

[[,] MAX_EVENT_SIZE = **size** [KB | MB]]

[[,] MEMORY_PARTITION_MODE = { NONE | PER_NODE | PER_CPU }]

[[,] TRACK_CAUSALITY = { ON | OFF }]

[[,] STARTUP_STATE = { ON | OFF }]

}

Arguments

event_session_name

Is the user-defined name for the event session. event_session_name is alphanumeric, can be up to 128 characters, must be unique within an instance of SQL Server, and must comply with the rules for Identifiers.

ADD EVENT [event_module_guid].event_package_name.event_name

Is the event to associate with the event session, where:

- event_module_guid is the GUID for the module that contains the event.
- event_package_name is the package that contains the action object.
- event_name is the event object.

Events appear in the sys.dm_xe_objects view as object_type 'event'.

SET { event_customizable_attribute = <value> [,...n] }

Allows customizable attributes for the event to be set. Customizable attributes appear in the sys.dm_xe_object_columns view as column_type 'customizable ' and object_name = event_name.

ACTION ({ [event_module_guid].event_package_name.action_name [,...n] })

Is the action to associate with the event session, where:

- event_module_guid is the GUID for the module that contains the event.
- event_package_name is the package that contains the action object.
- action_name is the action object.

Actions appear in the sys.dm_xe_objects view as object_type 'action'.

WHERE <predicate_expression>

Specifies the predicate expression used to determine if an event should be processed. If <predicate_expression> is true, the event is processed further by the actions and targets for the session. If <predicate_expression> is false, the event is dropped by the session before being processed by the actions and targets for the session. Predicate expressions are limited to 3000 characters, which limits string arguments.

event_field_name

Is the name of the event field that identifies the predicate source.

[event_module_guid].event_package_name.predicate_source_name

Is the name of the global predicate source where:

- event_module_guid is the GUID for the module that contains the event.
- event_package_name is the package that contains the predicate object.
- predicate_source_name is defined in the sys.dm_xe_objects view as object_type 'pred_source'.

[event_module_guid].event_package_name.predicate_compare_name

Is the name of the predicate object to associate with the event, where:

- event_module_guid is the GUID for the module that contains the event.
- event_package_name is the package that contains the predicate object.
- predicate_compare_name is a global source defined in the sys.dm_xe_objects view as object_type 'pred_compare'.

number

Is any numeric type including **decimal**. Limitations are the lack of available physical memory or a number that is too large to be represented as a 64-bit integer.

'string'

Either an ANSI or Unicode string as required by the predicate compare. No implicit string type conversion is performed for the predicate compare functions. Passing the wrong type results in an error.

ADD TARGET [event_module_guid].event_package_name.target_name

Is the target to associate with the event session, where:

- event_module_guid is the GUID for the module that contains the event.
- event_package_name is the package that contains the action object.
- target_name is the target. Targets appear in sys.dm_xe_objects view as object_type 'target'.

SET { target_parameter_name = <value> [, ...n] }

Sets a target parameter. Target parameters appear in the sys.dm_xe_object_columns view as column_type 'customizable' and object_name = *target_name*.

Important

If you are using the ring buffer target, we recommend that you set the max_memory target parameter to 2048 kilobytes (KB) to help avoid possible data truncation of the XML output. For more information about when to use the different target types, see SQL Server Extended Events Targets.

WITH (<event_session_options> [,...n])

Specifies options to use with the event session.

MAX_MEMORY = size [KB | MB]

Specifies the maximum amount of memory to allocate to the session for event buffering. The default is 4 MB. size is a whole number and can be a kilobyte (KB) or a megabyte (MB) value.

EVENT_RETENTION_MODE = { ALLOW_SINGLE_EVENT_LOSS | ALLOW_MULTIPLE_EVENT_LOSS | NO_EVENT_LOSS }

Specifies the event retention mode to use for handling event loss.

ALLOW_SINGLE_EVENT_LOSS

An event can be lost from the session. A single event is only dropped when all the event buffers are full. Losing a single event when event buffers are full allows for acceptable SQL Server performance characteristics, while minimizing the loss of data in the processed

event stream.

ALLOW_MULTIPLE_EVENT_LOSS

Full event buffers containing multiple events can be lost from the session. The number of events lost is dependant upon the memory size allocated to the session, the partitioning of the memory, and the size of the events in the buffer. This option minimizes performance impact on the server when event buffers are quickly filled, but large numbers of events can be lost from the session.

NO_EVENT_LOSS

No event loss is allowed. This option ensures that all events raised will be retained. Using this option forces all tasks that fire events to wait until space is available in an event buffer. This may cause detectable performance issues while the event session is active. User connections may stall while waiting for events to be flushed from the buffer.

MAX_DISPATCH_LATENCY = { seconds SECONDS | INFINITE }

Specifies the amount of time that events will be buffered in memory before being dispatched to event session targets. By default, this value is set to 30 seconds.

seconds SECONDS

The time, in seconds, to wait before starting to flush buffers to targets. seconds is a whole number. The minimum latency value is 1 second. However, 0 can be used to specify INFINITE latency.

INFINITE

Flush buffers to targets only when the buffers are full, or when the event session closes.

Note

MAX_DISPATCH_LATENCY = 0 SECONDS is equivalent to MAX_DISPATCH_LATENCY = INFINITE.

MAX_EVENT_SIZE = size [KB | MB]

Specifies the maximum allowable size for events. MAX_EVENT_SIZE should only be set to allow single events larger than MAX_MEMORY; setting it to less than MAX_MEMORY will raise an error. size is a whole number and can be a kilobyte (KB) or a megabyte (MB) value. If size is specified in kilobytes, the minimum allowable size is 64 KB. When MAX_EVENT_SIZE is set, two buffers of size are created in addition to MAX_MEMORY. This means that the total memory used for event buffering is MAX_MEMORY + 2 * MAX_EVENT_SIZE.

MEMORY_PARTITION_MODE = { NONE | PER_NODE | PER_CPU }

Specifies the location where event buffers are created.

NONE

A single set of buffers are created within the SQL Server instance.

PER_NODE

A set of buffers are created for each NUMA node.

PER_CPU

A set of buffers are created for each CPU.

TRACK_CAUSALITY = { ON | OFF }

Specifies whether or not causality is tracked. If enabled, causality allows related events on different server connections to be correlated together.

STARTUP_STATE = { ON | OFF }

Specifies whether or not to start this event session automatically when SQL Server starts.

Note

- If STARTUP_STATE = ON, the event session will only start if SQL Server is stopped and then restarted.

ON

The event session is started at startup.

OFF

The event session is not started at startup.

Remarks

The order of precedence for the logical operators is NOT (highest), followed by AND, followed by OR.

Permissions

Requires the ALTER ANY EVENT SESSION permission.

Examples

The following example shows how to create an event session named `test_session`. This example adds two events and uses the Event Tracing for Windows target.

```
IF EXISTS(SELECT * FROM sys.server_event_sessions WHERE name='test_session')
    DROP EVENT session test_session ON SERVER;
GO
CREATE EVENT SESSION test_session
ON SERVER
    ADD EVENT sqlos.async_io_requested,
    ADD EVENT sqlserver.lock_acquired
    ADD TARGET package0.etw_classic_sync_target
        (SET default_etw_session_logfile_path = N'C:\demo\traces\sqletw.etl'
)
    WITH (MAX_MEMORY=4MB, MAX_EVENT_SIZE=4MB);
GO
```

See Also

ALTER EVENT SESSION (Transact-SQL)

DROP EVENT SESSION (Transact-SQL)

sys.server_event_sessions

sys.dm_xe_objects

sys.dm_xe_object_columns

CREATE FULLTEXT CATALOG

Creates a full-text catalog for a database. One full-text catalog can have several full-text indexes, but a full-text index can only be part of one full-text catalog. Each database can contain zero or more full-text catalogs.

You cannot create full-text catalogs in the **master**, **model**, or **tempdb** databases.

Important

Beginning with SQL Server 2008, a full-text catalog is a virtual object and does not belong to any filegroup. A full-text catalog is a logical concept that refers to a group of full-text indexes.

Transact-SQL Syntax Conventions

Syntax

```
CREATE FULLTEXT CATALOG catalog_name
    [ON FILEGROUP filegroup ]
    [IN PATH 'rootpath']
    [WITH <catalog_option>]
    [AS DEFAULT]
    [AUTHORIZATION owner_name ]

<catalog_option>::=
    ACCENT_SENSITIVITY = {ON|OFF}
```

Arguments

catalog_name

Is the name of the new catalog. The catalog name must be unique among all catalog names in the current database. Also, the name of the file that corresponds to the full-text catalog (see ON FILEGROUP) must be unique among all files in the database. If the name of the catalog is already used for another catalog in the database, SQL Server returns an error.

The length of the catalog name cannot exceed 120 characters.

ON FILEGROUP filegroup

Beginning with SQL Server 2008, this clause has no effect.

IN PATH 'rootpath'

Note

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

Beginning with SQL Server 2008, this clause has no effect.

ACCENT_SENSITIVITY = {ON|OFF}

Specifies that the catalog is accent sensitive or accent insensitive for full-text indexing. When this property is changed, the index must be rebuilt. The default is to use the accent-sensitivity specified in the database collation. To display the database collation, use the **sys.databases** catalog view.

To determine the current accent-sensitivity property setting of a full-text catalog, use the FULLTEXTCATALOGPROPERTY function with the **accentsensitivity** property value against catalog_name. If the value returned is '1', the full-text catalog is accent sensitive; if the value is '0', the catalog is not accent-sensitive.

AS DEFAULT

Specifies that the catalog is the default catalog. When full-text indexes are created without a full-text catalog explicitly specified, the default catalog is used. If an existing full-text catalog is already marked AS DEFAULT, setting this new catalog AS DEFAULT will make this catalog the default full-text catalog.

AUTHORIZATION owner_name

Sets the owner of the full-text catalog to the name of a database user or role. If owner_name is a role, the role must be the name of a role that the current user is a member of, or the user running the statement must be the database owner or system administrator.

If owner_name is a user name, the user name must be one of the following:

- The name of the user running the statement.
- The name of a user that the user executing the command has impersonate permissions for.
- Or, the user executing the command must be the database owner or system administrator.

owner_name must also be granted TAKE OWNERSHIP permission on the specified full-text catalog.

Remarks

Full-text catalog IDs start at 00005 and are incremented by one for each new catalog created.

Permissions

User must have CREATE FULLTEXT CATALOG permission on the database, or be a member of the **db_owner**, or **db_ddladmin** fixed database roles.

Examples

The following example creates a full-text catalog and also a full-text index.

```
USE AdventureWorks;
GO
CREATE FULLTEXT CATALOG ftCatalog AS DEFAULT;
GO
CREATE FULLTEXT INDEX ON HumanResources.JobCandidate(Resume) KEY INDEX
PK_JobCandidate_JobCandidateID;
GO
```

See Also

sys.fulltext_catalogs (Transact-SQL)
New Full-Text Catalog (General Page)
DROP FULLTEXT CATALOG
Full-Text Search
New Full-Text Catalog (General Page)

CREATE FULLTEXT INDEX

Creates a full-text index on a table or indexed view in a database. Only one full-text index is allowed per table or indexed view, and each full-text index applies to a single table or indexed view.

A full-text index can contain up to 1024 columns.

 Transact-SQL Syntax Conventions

Syntax

```
CREATE FULLTEXT INDEX ON table_name
  [ ( { column_name
        [ TYPE COLUMN type_column_name ]
        [ LANGUAGE language_term ]
        [ STATISTICAL_SEMANTICS ]
      } [ ,...n]
    ) ]
```

```
   KEY INDEX index_name
   [ ON <catalog_filegroup_option> ]
   [ WITH [ ( ] <with_option> [ ,...n] [ ) ] ]
[;]

<catalog_filegroup_option>::=
{
   fulltext_catalog_name
| ( fulltext_catalog_name, FILEGROUP filegroup_name )
| ( FILEGROUP filegroup_name, fulltext_catalog_name )
| ( FILEGROUP filegroup_name )
}

<with_option>::=
{
   CHANGE_TRACKING [ = ] { MANUAL | AUTO | OFF [, NO POPULATION ] }
| STOPLIST [ = ] { OFF | SYSTEM | stoplist_name }
| SEARCH PROPERTY LIST [ = ] property_list_name
}
```

Arguments

table_name

Is the name of the table or indexed view that contains the column or columns included in the full-text index.

column_name

Is the name of the column included in the full-text index. Only columns of type **char**, **varchar**, **nchar**, **nvarchar**, **text**, **ntext**, **image**, **xml**, and **varbinary(max)** can be indexed for full-text search. To specify multiple columns, repeat the column_name clause as follows:

CREATE FULLTEXT INDEX ON table_name (column_name1 [...], column_name2 [...]) ...

TYPE COLUMN type_column_name

Specifies the name of a table column, type_column_name, that is used to hold the document type for a **varbinary(max)** or **image** document. This column, known as the type column, contains a user-supplied file extension (.doc, .pdf, .xls, and so forth). The type column must be of type **char**, **nchar**, **varchar**, or **nvarchar**.

Specify TYPE COLUMN type_column_name only if column_name specifies a **varbinary(max)** or **image** column, in which data is stored as binary data; otherwise, SQL Server returns an error.

Note

At indexing time, the Full-Text Engine uses the abbreviation in the type column of each table row to identify which full-text search filter to use for the document in column_name. The filter loads the document as a binary stream, removes the formatting information, and sends the text from the document to the word-breaker component. For more information, see Full-Text Search Filters.

LANGUAGE language_term

Is the language of the data stored in column_name.

language_term is optional and can be specified as a string, integer, or hexadecimal value corresponding to the locale identifier (LCID) of a language. If no value is specified, the default language of the SQL Server instance is used.

If language_term is specified, the language it represents will be used to index data stored in **char**, **nchar**, **varchar**, **nvarchar**, **text**, and **ntext** columns. This language is the default language used at query time if language_term is not specified as part of a full-text predicate against the column.

When specified as a string, language_term corresponds to the alias column value in the syslanguages system table. The string must be enclosed in single quotation marks, as in 'language_term'. When specified as an integer, language_term is the actual LCID that identifies the language. When specified as a hexadecimal value, language_term is 0x followed by the hex value of the LCID. The hex value must not exceed eight digits, including leading zeros.

If the value is in double-byte character set (DBCS) format, SQL Server will convert it to Unicode.

Resources, such as word breakers and stemmers, must be enabled for the language specified as language_term. If such resources do not support the specified language, SQL Server returns an error.

Use the sp_configure stored procedure to access information about the default full-text language of the Microsoft SQL Server instance. For more information, see sp_configure (Transact-SQL).

For non-BLOB and non-XML columns containing text data in multiple languages, or for cases when the language of the text stored in the column is unknown, it might be appropriate for you to use the neutral (0x0) language resource. However, first you should understand the possible consequences of using the neutral (0x0) language resource. For information about the possible solutions and consequences of using the neutral (0x0) language resource, see International Considerations for Full-Text Search.

For documents stored in XML- or BLOB-type columns, the language encoding within the document will be used at indexing time. For example, in XML columns, the **xml:lang** attribute in XML documents will identify the language. At query time, the value previously specified in language_term becomes the default language used for full-text queries unless language_term is specified as part of a full-text query.

STATISTICAL_SEMANTICS

Creates the additional key phrase and document similarity indexes that are part of statistical semantic indexing. For more information, see Semantic Search.

KEY INDEX index_name

Is the name of the unique key index on table_name. The KEY INDEX must be a unique, single-key, non-nullable column. Select the smallest unique key index for the full-text unique key. For the best performance, we recommend an integer data type for the full-text key.

fulltext_catalog_name

Is the full-text catalog used for the full-text index. The catalog must already exist in the database. This clause is optional. If it is not specified, a default catalog is used. If no default catalog exists, SQL Server returns an error.

FILEGROUP filegroup_name

Creates the specified full-text index on the specified filegroup. The filegroup must already exist. If the FILEGROUP clause is not specified, the full-text index is placed in the same filegroup as base table or view for a nonpartitioned table or in the primary filegroup for a partitioned table.

CHANGE_TRACKING [=] { MANUAL | AUTO | OFF [, NO POPULATION] }

Specifies whether changes (updates, deletes or inserts) made to table columns that are covered by the full-text index will be propagated by SQL Server to the full-text index. Data changes through WRITETEXT and UPDATETEXT are not reflected in the full-text index, and are not picked up with change tracking.

MANUAL

Specifies that the tracked changes must be propagated manually by calling the ALTER FULLTEXT INDEX ... START UPDATE POPULATION Transact-SQL statement (*manual population*). You can use SQL Server Agent to call this Transact-SQL statement periodically.

AUTO

Specifies that the tracked changes will be propagated automatically as data is modified in the base table (*automatic population*). Although changes are propagated automatically, these changes might not be reflected immediately in the full-text index. AUTO is the default.

OFF [, NO POPULATION]

Specifies that SQL Server does not keep a list of changes to the indexed data. When NO POPULATION is not specified, SQL Server populates the index fully after it is created.

The NO POPULATION option can be used only when CHANGE_TRACKING is OFF. When NO POPULATION is specified, SQL Server does not populate an index after it is created. The index is only populated after the user executes the ALTER FULLTEXT INDEX command with the START FULL POPULATION or START INCREMENTAL POPULATION clause.

STOPLIST [=] { OFF | SYSTEM | stoplist_name }

Associates a full-text stoplist with the index. The index is not populated with any tokens that are part of the specified stoplist. If STOPLIST is not specified, SQL Server associates the system full-text stoplist with the index.

OFF

Specifies that no stoplist be associated with the full-text index.

SYSTEM

Specifies that the default full-text system STOPLIST should be used for this full-text index.

stoplist_name

Specifies the name of the stoplist to be associated with the full-text index.

SEARCH PROPERTY LIST [=] property_list_name

Associates a search property list with the index.

OFF

Specifies that no property list be associated with the full-text index.

property_list_name

Specifies the name of the search property list to associate with the full-text index.

Remarks

For more information about full-text indexes, see Create and Manage Full-Text Indexes.

On **xml** columns, you can create a full-text index that indexes the content of the XML elements, but ignores the XML markup. Attribute values are full-text indexed unless they are numeric values. Element tags are used as token boundaries. Well-formed XML or HTML documents and fragments containing multiple languages are supported. For more information, see Full-Text Index on an XML Column.

We recommend that the index key column is an integer data type. This provides optimizations at query execution time.

Interactions of Change Tracking and NO POPULATION Parameter

Whether the full-text index is populated depends on whether change-tracking is enabled and whether WITH NO POPULATION is specified in the ALTER FULLTEXT INDEX statement. The following table summarizes the result of their interaction.

Change Tracking	WITH NO POPULATION	Result
Not Enabled	Not specified	A full population is performed on the index.
Not Enabled	Specified	No population of the index occurs until an ALTER FULLTEXT

Change Tracking	WITH NO POPULATION	Result
		INDEX...START POPULATION statement is issued.
Enabled	Specified	An error is raised, and the index is not altered.
Enabled	Not specified	A full population is performed on the index.

For more information about populating full-text indexes, see Full-Text Index Population.

Permissions

User must have REFERENCES permission on the full-text catalog and have ALTER permission on the table or indexed view, or be a member of the sysadmin fixed server role, or db_owner, or db_ddladmin fixed database roles.

If SET STOPLIST is specified, the user must have REFERENCES permission on the specified stoplist. The owner of the STOPLIST can grant this permission.

Note
The public is granted REFERENCE permission to the default stoplist that is shipped with SQL Server.

Examples

A. Creating a unique index, a full-text catalog, and a full-text index

The following example creates a unique index on the `JobCandidateID` column of the `HumanResources.JobCandidate` table of the `AdventureWorks` sample database. The example then creates a default full-text catalog, `ft`. Finally, the example creates a full-text index on the `Resume` column, using the `ft` catalog and the system stoplist.

```
USE AdventureWorks;
GO
CREATE UNIQUE INDEX ui_ukJobCand ON
HumanResources.JobCandidate(JobCandidateID);
CREATE FULLTEXT CATALOG ft AS DEFAULT;
CREATE FULLTEXT INDEX ON HumanResources.JobCandidate(Resume)
   KEY INDEX ui_ukJobCand
   WITH STOPLIST = SYSTEM;
GO
```

B. Creating a full-text index on several table columns

The following example creates a full-text catalog, `production_catalog`, in the `AdventureWorks` sample database. The example then creates a full-text index that uses this new catalog. The full-text index is on the on the `ReviewerName`, `EmailAddress`, and `Comments` columns of the `Production.ProductReview` table of the `AdventureWorks` sample database. For each column, the example specifies the LCID of English, `1033`, which is the language of the data in the columns. This full-text index uses an existing unique key index, `PK_ProductReview_ProductReviewID`. As recommended, this index key is on an integer column, `ProductReviewID`.

```
USE AdventureWorks;
GO
CREATE FULLTEXT CATALOG production_catalog;
GO
CREATE FULLTEXT INDEX ON Production.ProductReview
 (
  ReviewerName
     Language 1033,
  EmailAddress
     Language 1033,
  Comments
     Language 1033
 )
  KEY INDEX PK_ProductReview_ProductReviewID
      ON production_catalog;
GO
```

C. Creating a full-text index with a search property list without populating it

The following example creates a full-text index on the `Title`, `DocumentSummary`, and `Document` columns of the `Production.Document` table. The example specifies the LCID of English, `1033`, which is the language of the data in the columns. This full-text index uses the default full-text catalog and an existing unique key index, `PK_Document_DocumentID`. As recommended, this index key is on an integer column, `DocumentID`.

The example specifies the SYSTEM stoplist. It also specifies a search property list, `DocumentPropertyList`; for an example that creates this property list, see CREATE SEARCH PROPERTY LIST (Transact-SQL).

The example specifies that change tracking is off with no population. Later, during off-peak hours, the example uses an ALTER FULLTEXT INDEX statement to start a full population on the new index and enable automatic change tracking.

```
USE AdventureWorks;
```

```
GO
CREATE FULLTEXT INDEX ON Production.Document
  (
  Title
      Language 1033,
  DocumentSummary
      Language 1033,
  Document
      TYPE COLUMN FileExtension
      Language 1033
  )
  KEY INDEX PK_Document_DocumentID
         WITH STOPLIST = SYSTEM, SEARCH PROPERTY LIST =
DocumentPropertyList, CHANGE_TRACKING OFF, NO POPULATION;
   GO
```

Later, at an off-peak time, the index is populated:

```
ALTER FULLTEXT INDEX ON Production.Document SET CHANGE_TRACKING AUTO;
GO
```

See Also

Create and Manage Full-Text Indexes
ALTER FULLTEXT INDEX
DROP FULLTEXT INDEX
Full-Text Search
GRANT
sys.fulltext_indexes (Transact-SQL)
Using Property Lists to Search for Document Properties

CREATE FULLTEXT STOPLIST

Creates a new full-text stoplist in the current database.

In SQL Server 2008 and later versions, stopwords are managed in databases by using objects called *stoplists*. A stoplist is a list of stopwords that, when associated with a full-text index, is applied to full-text queries on that index. For more information, see Stopwords and Stoplists.

Important
CREATE FULLTEXT STOPLIST, ALTER FULLTEXT STOPLIST, and DROP FULLTEXT STOPLIST are supported only under compatibility level 100. Under compatibility levels 80 and 90, these statements are not supported. However, under all compatibility levels the system stoplist is automatically associated with new full-text indexes.

Transact-SQL Syntax Conventions

Syntax

```
CREATE FULLTEXT STOPLIST stoplist_name
[ FROM { [ database_name.]source_stoplist_name } | SYSTEM STOPLIST ]
[ AUTHORIZATION owner_name ]
;
```

Arguments

stoplist_name

Is the name of the stoplist. stoplist_name can be a maximum of 128 characters. stoplist_name must be unique among all stoplists in the current database, and conform to the rules for identifiers.

stoplist_name will be used when the full-text index is created.

database_name

Is the name of the database where the stoplist specified by source_stoplist_name is located. If not specified, database_name defaults to the current database.

source_stoplist_name

Specifies that the new stoplist is created by copying an existing stoplist. If source_stoplist_name does not exist, or the database user does not have correct permissions, CREATE FULLTEXT STOPLIST fails with an error. If any languages specified in the stop words of the source stoplist are not registered in the current database, CREATE FULLTEXT STOPLIST succeeds, but warning(s) are returned and the corresponding stop words are not added.

SYSTEM STOPLIST

Specifies that the new stoplist is created from the stoplist that exists by default in the Resource database.

AUTHORIZATION owner_name

Specifies the name of a database principal to own of the stoplist. owner_name must either be the name of a principal of which the current user is a member, or the current user must have IMPERSONATE permission on owner_name. If not specified, ownership is given to the current user.

Remarks

The creator of a stoplist is its owner.

Permissions

To create a STOPLIST requires CREATE FULLTEXT CATALOG permissions. The stoplist owner can grant CONTROL permission explicitly on a stoplist to allow users to add and remove words and to drop the stoplist.

Note
Using a stoplist with a full-text index requires REFERENCE permission.

Examples

A. Creating a new full-text stoplist

The following example creates a new full-text stoplist named `myStoplist`.

```
CREATE FULLTEXT STOPLIST myStoplist;
GO
```

B. Copying a full-text stoplist from an existing full-text stoplist

The following example creates a new full-text stoplist named `myStoplist2` by copying an existing AdventureWorks stoplist named `Customers.otherStoplist`.

```
CREATE FULLTEXT STOPLIST myStoplist2 FROM AdventureWorks.otherStoplist;
GO
```

C. Copying a full-text stoplist from the system full-text stoplist

The following example creates a new full-text stoplist named `myStoplist3` by copying from the system stoplist.

```
CREATE FULLTEXT STOPLIST myStoplist3 FROM SYSTEM STOPLIST;
GO
```

See Also

ALTER FULLTEXT STOPLIST (Transact-SQL)
DROP FULLTEXT STOPLIST (Transact-SQL)
Noise Words
sys.fulltext_stoplists (Transact-SQL)
sys.fulltext_stopwords (Transact-SQL)
Configure and Manage Stopwords and Stoplists for Full-Text Search

CREATE FUNCTION

Creates a user-defined function in SQL Server 2012. A user-defined function is a Transact-SQL or common language runtime (CLR) routine that accepts parameters, performs an action, such as a complex calculation, and returns the result of that action as a value. The return value can either

be a scalar (single) value or a table. Use this statement to create a reusable routine that can be used in these ways:

- In Transact-SQL statements such as SELECT
- In applications calling the function
- In the definition of another user-defined function
- To parameterize a view or improve the functionality of an indexed view
- To define a column in a table
- To define a CHECK constraint on a column
- To replace a stored procedure

 Transact-SQL Syntax Conventions

Syntax

```
--Transact-SQL Scalar Function Syntax
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ][ type_schema_name. ] parameter_data_type
    [ = default ] [ READONLY ] }
    [ ,...n ]
  ]
)
RETURNS return_data_type
    [ WITH <function_option> [ ,...n ] ]
    [ AS ]
    BEGIN
        function_body
        RETURN scalar_expression
    END
[ ; ]
--Transact-SQL Inline Table-Valued Function Syntax
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ] [ type_schema_name. ] parameter_data_type
    [ = default ] [ READONLY ] }
    [ ,...n ]
  ]
)
RETURNS TABLE
    [ WITH <function_option> [ ,...n ] ]
```

```
   [ AS ]
   RETURN [ ( ] select_stmt [ ) ]
[ ; ]

--Transact-SQL Multistatement Table-valued Function Syntax
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ] [ type_schema_name. ] parameter_data_type
   [ = default ] [READONLY] }
   [ ,...n ]
 ]
)
RETURNS @return_variable TABLE <table_type_definition>
   [ WITH <function_option> [ ,...n ] ]
   [ AS ]
   BEGIN
      function_body
      RETURN
   END
[ ; ]

--Transact-SQL Function Clauses
<function_option>::=
{
   [ ENCRYPTION ]
  | [ SCHEMABINDING ]
  | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
  | [ EXECUTE_AS_Clause ]
}

<table_type_definition>:: =
( { <column_definition> <column_constraint>
  | <computed_column_definition> }
   [ <table_constraint> ] [ ,...n ]
)
```

```
<column_definition>::=
{
   { column_name data_type }
   [ [ DEFAULT constant_expression ]
     [ COLLATE collation_name ] | [ ROWGUIDCOL ]
   ]
   | [ IDENTITY [ (seed , increment ) ] ]
   [ <column_constraint> [ ...n ] ]
}

<column_constraint>::=
{
   [ NULL | NOT NULL ]
   { PRIMARY KEY | UNIQUE }
     [ CLUSTERED | NONCLUSTERED ]
     [ WITH FILLFACTOR = fillfactor
       | WITH ( < index_option > [ , ...n ] )
     [ ON { filegroup | "default" } ]
 | [ CHECK ( logical_expression ) ] [ ,...n ]
}

<computed_column_definition>::=
column_name AS computed_column_expression

<table_constraint>::=
{
   { PRIMARY KEY | UNIQUE }
     [ CLUSTERED | NONCLUSTERED ]
     ( column_name [ ASC | DESC ] [ ,...n ] )
       [ WITH FILLFACTOR = fillfactor
       | WITH ( <index_option> [ , ...n ] )
 | [ CHECK ( logical_expression ) ] [ ,...n ]
}

<index_option>::=
```

```
{
    PAD_INDEX = { ON | OFF }
  | FILLFACTOR = fillfactor
  | IGNORE_DUP_KEY =  { ON | OFF }
  | STATISTICS_NORECOMPUTE = { ON | OFF }
  | ALLOW_ROW_LOCKS = { ON | OFF }
  | ALLOW_PAGE_LOCKS ={ ON | OFF }
}
--CLR Scalar Function Syntax
CREATE FUNCTION [ schema_name. ] function_name
( { @parameter_name [AS] [ type_schema_name. ] parameter_data_type
    [ = default ] }
    [ ,...n ]
)
RETURNS { return_data_type }
    [ WITH <clr_function_option> [ ,...n ] ]
    [ AS ] EXTERNAL NAME <method_specifier>
[ ; ]
--CLR Table-Valued Function Syntax
CREATE FUNCTION [ schema_name. ] function_name
( { @parameter_name [AS] [ type_schema_name. ] parameter_data_type
    [ = default ] }
    [ ,...n ]
)
RETURNS TABLE <clr_table_type_definition>
    [ WITH <clr_function_option> [ ,...n ] ]
    [ ORDER ( <order_clause> ) ]
    [ AS ] EXTERNAL NAME <method_specifier>
[ ; ]
--CLR Function Clauses
<order_clause> ::=
{
  <column_name_in_clr_table_type_definition>
  [ ASC | DESC ]
} [ ,...n]
```

<method_specifier>::=

```
    assembly_name.class_name.method_name
```

<clr_function_option>::=

}

[RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT]

| [**EXECUTE_AS_Clause**]

}

<clr_table_type_definition>::=

({ **column_name data_type** } [**,**...**n**] **)**

Arguments

schema_name

Is the name of the schema to which the user-defined function belongs.

function_name

Is the name of the user-defined function. Function names must comply with the rules for identifiers and must be unique within the database and to its schema.

Note

Parentheses are required after the function name even if a parameter is not specified.

@parameter_name

Is a parameter in the user-defined function. One or more parameters can be declared.

A function can have a maximum of 2,100 parameters. The value of each declared parameter must be supplied by the user when the function is executed, unless a default for the parameter is defined.

Specify a parameter name by using an at sign (@) as the first character. The parameter name must comply with the rules for identifiers. Parameters are local to the function; the same parameter names can be used in other functions. Parameters can take the place only of constants; they cannot be used instead of table names, column names, or the names of other database objects.

Note

ANSI_WARNINGS is not honored when you pass parameters in a stored procedure, user-defined function, or when you declare and set 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.

[type_schema_name.] parameter_data_type

Is the parameter data type, and optionally the schema to which it belongs. For Transact-SQL functions, all data types, including CLR user-defined types and user-defined table types, are allowed except the **timestamp** data type. For CLR functions, all data types, including CLR user-defined types, are allowed except **text**, **ntext**, **image**, user-defined table types and **timestamp** data types. The nonscalar types, **cursor** and **table**, cannot be specified as a parameter data type in either Transact-SQL or CLR functions.

If type_schema_name is not specified, the Database Engine looks for the scalar_parameter_data_type in the following order:

- The schema that contains the names of SQL Server system data types.
- The default schema of the current user in the current database.
- The **dbo** schema in the current database.

[= default]

Is a default value for the parameter. If a default value is defined, the function can be executed without specifying a value for that parameter.

Note

Default parameter values can be specified for CLR functions except for the **varchar(max)** and **varbinary(max)** data types.

When a parameter of the function has a default value, the keyword DEFAULT must be specified when the function is called to retrieve the default value. This behavior is different from using parameters with default values in stored procedures in which omitting the parameter also implies the default value. However, the DEFAULT keyword is not required when invoking a scalar function by using the EXECUTE statement.

READONLY

Indicates that the parameter cannot be updated or modified within the definition of the function. If the parameter type is a user-defined table type, READONLY should be specified.

return_data_type

Is the return value of a scalar user-defined function. For Transact-SQL functions, all data types, including CLR user-defined types, are allowed except the **timestamp** data type. For CLR functions, all data types, including CLR user-defined types, are allowed except the **text**, **ntext**, **image**, and **timestamp** data types. The nonscalar types, **cursor** and **table**, cannot be specified as a return data type in either Transact-SQL or CLR functions.

function_body

Specifies that a series of Transact-SQL statements, which together do not produce a side effect such as modifying a table, define the value of the function. function_body is used only in scalar functions and multistatement table-valued functions.

In scalar functions, function_body is a series of Transact-SQL statements that together evaluate to a scalar value.

In multistatement table-valued functions, function_body is a series of Transact-SQL statements that populate a TABLE return variable.

scalar_expression

Specifies the scalar value that the scalar function returns.

TABLE

Specifies that the return value of the table-valued function is a table. Only constants and @local_variables can be passed to table-valued functions.

In inline table-valued functions, the TABLE return value is defined through a single SELECT statement. Inline functions do not have associated return variables.

In multistatement table-valued functions, @return_variable is a TABLE variable, used to store and accumulate the rows that should be returned as the value of the function. @return_variable can be specified only for Transact-SQL functions and not for CLR functions.

Warning

Joining to a multistatement table valued function in a **FROM** clause is possible, but can give poor performance. SQL Server is unable to use all the optimized techniques against some statements that can be included in a multistatement function, resulting in a suboptimal query plan. To obtain the best possible performance, whenever possible use joins between base tables instead of functions.

select_stmt

Is the single SELECT statement that defines the return value of an inline table-valued function.

ORDER (<order_clause>)

Specifies the order in which results are being returned from the table-valued function. For more information, see the section, "Guidance on Using Sort Order," later in this topic.

EXTERNAL NAME <method_specifier> assembly_name.class_name.method_name

Specifies the method of an assembly to bind with the function. assembly_name must match an existing assembly in SQL Server in the current database with visibility on. 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 (" "). method_name must be a valid SQL Server identifier and must exist as a static method in the specified class.

Note

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

Note

This option is not available in a contained database.

<table_type_definition> ({ <column_definition> <column_constraint> | <computed_column_definition> } [<table_constraint>] [,...n])

Defines the table data type for a Transact-SQL function. The table declaration includes column definitions and column or table constraints. The table is always put in the primary filegroup.

< clr_table_type_definition > ({ column_name data_type } [,...n])

Defines the table data types for a CLR function. The table declaration includes only column names and data types. The table is always put in the primary filegroup.

<function_option>::= and <clr_function_option>::=

Specifies that the function will have one or more of the following options.

ENCRYPTION

Indicates that the Database Engine will convert the original text of the CREATE FUNCTION statement to an obfuscated format. The output of the obfuscation is not directly visible in any catalog views. Users that have no access to system tables or database files cannot retrieve the obfuscated text. However, the text will be available to privileged users that can either access system tables over the DAC port or directly access database files. Also, users that can attach a debugger to the server process can retrieve the original procedure from memory at runtime. For more information about accessing system metadata, see CLR User-Defined Functions.

Using this option prevents the function from being published as part of SQL Server replication. This option cannot be specified for CLR functions.

SCHEMABINDING

Specifies that the function is bound to the database objects that it references. When SCHEMABINDING is specified, the base objects cannot be modified in a way that would affect the function definition. The function definition itself must first be modified or dropped to remove dependencies on the object that is to be modified.

The binding of the function to the objects it references is removed only when one of the following actions occurs:

- The function is dropped.
- The function is modified by using the ALTER statement with the SCHEMABINDING option not specified.

A function can be schema bound only if the following conditions are true:

- The function is a Transact-SQL function.
- The user-defined functions and views referenced by the function are also schema-bound.
- The objects referenced by the function are referenced using a two-part name.
- The function and the objects it references belong to the same database.
- The user who executed the CREATE FUNCTION statement has REFERENCES permission

on the database objects that the function references.

RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT

Specifies the **OnNULLCall** attribute of a scalar-valued function. If not specified, CALLED ON NULL INPUT is implied by default. This means that the function body executes even if NULL is passed as an argument.

If RETURNS NULL ON NULL INPUT is specified in a CLR function, it indicates that SQL Server can return NULL when any of the arguments it receives is NULL, without actually invoking the body of the function. If the method of a CLR function specified in <method_specifier> already has a custom attribute that indicates RETURNS NULL ON NULL INPUT, but the CREATE FUNCTION statement indicates CALLED ON NULL INPUT, the CREATE FUNCTION statement takes precedence. The **OnNULLCall** attribute cannot be specified for CLR table-valued functions.

EXECUTE AS Clause

Specifies the security context under which the user-defined function is executed. Therefore, you can control which user account SQL Server uses to validate permissions on any database objects that are referenced by the function.

Note

EXECUTE AS cannot be specified for inline user-defined functions.

For more information, see EXECUTE AS Clause (Transact-SQL).

< column_definition >::=

Defines the table data type. The table declaration includes column definitions and constraints. For CLR functions, only column_name and data_type can be specified.

column_name

Is the name of a column in the table. Column names must comply with the rules for identifiers and must be unique in the table. column_name can consist of 1 through 128 characters.

data_type

Specifies the column data type. For Transact-SQL functions, all data types, including CLR user-defined types, are allowed except **timestamp**. For CLR functions, all data types, including CLR user-defined types, are allowed except **text**, **ntext**, **image**, **char**, **varchar**, **varchar(max)**, and **timestamp**.The nonscalar type **cursor** cannot be specified as a column data type in either Transact-SQL or CLR functions.

DEFAULT constant_expression

Specifies the value provided for the column when a value is not explicitly supplied during an insert. constant_expression is a constant, NULL, or a system function value. DEFAULT definitions can be applied to any column except those that have the IDENTITY property. DEFAULT cannot be specified for CLR table-valued functions.

COLLATE collation_name

Specifies the collation for the column. If not specified, the column is assigned the default collation of the database. Collation name can be either a Windows collation name or a SQL collation name. For a list of and more information about collations, see Windows Collation Name and SQL Collation Name.

The COLLATE clause can be used to change the collations only of columns of the **char**, **varchar**, **nchar**, and **nvarchar** data types.

COLLATE cannot be specified for CLR table-valued functions.

ROWGUIDCOL

Indicates that the new column is a row globally unique identifier column. Only one **uniqueidentifier** column per table can be designated as the ROWGUIDCOL column. The ROWGUIDCOL property can be assigned only to a **uniqueidentifier** column.

The ROWGUIDCOL property does not enforce uniqueness of the values stored in the column. It also does not automatically generate values for new rows inserted into the table. To generate unique values for each column, use the NEWID function on INSERT statements. A default value can be specified; however, NEWID cannot be specified as the default.

IDENTITY

Indicates that the new column is an identity column. When a new row is added to the table, SQL Server provides a unique, incremental value for the column. Identity columns are typically used together with PRIMARY KEY constraints to serve as the unique row identifier for the table. The IDENTITY property can be assigned to **tinyint**, **smallint**, **int**, **bigint**, **decimal(p,0)**, or **numeric(p,0)** columns. Only one identity column can be created per table. Bound defaults and DEFAULT constraints cannot be used with an identity column. You must specify both the seed and increment or neither. If neither is specified, the default is (1,1).

IDENTITY cannot be specified for CLR table-valued functions.

seed

Is the integer value to be assigned to the first row in the table.

increment

Is the integer value to add to the seed value for successive rows in the table.

< column_constraint >::= and < table_constraint>::=

Defines the constraint for a specified column or table. For CLR functions, the only constraint type allowed is NULL. Named constraints are not allowed.

NULL | NOT NULL

Determines whether null values are allowed in the column. NULL is not strictly a constraint but can be specified just like NOT NULL. NOT NULL cannot be specified for CLR table-valued functions.

PRIMARY KEY

Is a constraint that enforces entity integrity for a specified column through a unique index. In

table-valued user-defined functions, the PRIMARY KEY constraint can be created on only one column per table. PRIMARY KEY cannot be specified for CLR table-valued functions.

UNIQUE

Is a constraint that provides entity integrity for a specified column or columns through a unique index. A table can have multiple UNIQUE constraints. UNIQUE cannot be specified for CLR table-valued functions.

CLUSTERED | NONCLUSTERED

Indicate that a clustered or a nonclustered index is created for the PRIMARY KEY or UNIQUE constraint. PRIMARY KEY constraints use CLUSTERED, and UNIQUE constraints use NONCLUSTERED.

CLUSTERED can be specified for only one constraint. If CLUSTERED is specified for a UNIQUE constraint and a PRIMARY KEY constraint is also specified, the PRIMARY KEY uses NONCLUSTERED.

CLUSTERED and NONCLUSTERED cannot be specified for CLR table-valued functions.

CHECK

Is a constraint that enforces domain integrity by limiting the possible values that can be entered into a column or columns. CHECK constraints cannot be specified for CLR table-valued functions.

logical_expression

Is a logical expression that returns TRUE or FALSE.

<computed_column_definition>::=

Specifies a computed column. For more information about computed columns, see CREATE TABLE (Transact-SQL).

column_name

Is the name of the computed column.

computed_column_expression

Is an expression that defines the value of a computed column.

<index_option>::=

Specifies the index options for the PRIMARY KEY or UNIQUE index. For more information about index options, see CREATE INDEX (Transact-SQL).

PAD_INDEX = { ON | OFF }

Specifies index padding. The default is OFF.

FILLFACTOR = fillfactor

Specifies a percentage that indicates how full the Database Engine should make the leaf level of each index page during index creation or change. fillfactor must be an integer value from 1 to 100. The default is 0.

IGNORE_DUP_KEY = { ON | OFF }

Specifies the error response when an insert operation attempts to insert duplicate key values into a unique index. The IGNORE_DUP_KEY option applies only to insert operations after the index is created or rebuilt. The default is OFF.

STATISTICS_NORECOMPUTE = { ON | OFF }

Specifies whether distribution statistics are recomputed. The default is OFF.

ALLOW_ROW_LOCKS = { ON | OFF }

Specifies whether row locks are allowed. The default is ON.

ALLOW_PAGE_LOCKS = { ON | OFF }

Specifies whether page locks are allowed. The default is ON.

Best Practices

If a user-defined function is not created with the SCHEMABINDING clause, changes that are made to underlying objects can affect the definition of the function and produce unexpected results when it is invoked. We recommend that you implement one of the following methods to ensure that the function does not become outdated because of changes to its underlying objects:

- Specify the WITH SCHEMABINDING clause when you are creating the function. This ensures that the objects referenced in the function definition cannot be modified unless the function is also modified.
- Execute the sp_refreshsqlmodule stored procedure after modifying any object that is specified in the definition of the function.

Data Types

If parameters are specified in a CLR function, they should be SQL Server types as defined previously for scalar_parameter_data_type. For information about comparing SQL Server system data types to CLR integration data types or .NET Framework common language runtime data types, see SQL Data Types and Their .NET Equivalents.

For SQL Server to reference the correct method when it is overloaded in a class, the method indicated in <method_specifier> must have the following characteristics:

- Receive the same number of parameters as specified in [,...n].
- Receive all the parameters by value, not by reference.
- Use parameter types that are compatible with those specified in the SQL Server function.

If the return data type of the CLR function specifies a table type (RETURNS TABLE), the return data type of the method in <method_specifier> should be of type **IEnumerator** or **IEnumerable**, and it is assumed that the interface is implemented by the creator of the function. Unlike Transact-SQL functions, CLR functions cannot include PRIMARY KEY, UNIQUE, or CHECK constraints in <table_type_definition>. The data types of columns specified in <table_type_definition> must match the types of the corresponding columns of the result set

returned by the method in <method_specifier> at execution time. This type-checking is not performed at the time the function is created.

For more information about how to program CLR functions, see CLR User-defined Functions.

General Remarks

Scalar-valued functions can be invoked where scalar expressions are used. This includes computed columns and CHECK constraint definitions. Scalar-valued functions can also be executed by using the EXECUTE statement. Scalar-valued functions must be invoked by using at least the two-part name of the function. For more information about multipart names, see Transact-SQL Syntax Conventions (Transact-SQL). Table-valued functions can be invoked where table expressions are allowed in the FROM clause of SELECT, INSERT, UPDATE, or DELETE statements. For more information, see Executing User-Defined Functions (Database Engine).

Interoperability

The following statements are valid in a function:

- Assignment statements.
- Control-of-Flow statements except TRY...CATCH statements.
- DECLARE statements defining local data variables and local cursors.
- SELECT statements that contain select lists with expressions that assign values to local variables.
- Cursor operations referencing local cursors that are declared, opened, closed, and deallocated in the function. Only FETCH statements that assign values to local variables using the INTO clause are allowed; FETCH statements that return data to the client are not allowed.
- INSERT, UPDATE, and DELETE statements modifying local table variables.
- EXECUTE statements calling extended stored procedures.
- For more information, see Creating User-Defined Functions (Database Engine).

Computed Column Interoperability

In SQL Server 2005 and later, functions have the following properties. The values of these properties determine whether functions can be used in computed columns that can be persisted or indexed.

Property	Description	Notes
IsDeterministic	Function is deterministic or nondeterministic.	Local data access is allowed in deterministic functions. For example, functions that always return the same result any time they are called by using a specific set of input

Property	Description	Notes
		values and with the same state of the database would be labeled deterministic.
IsPrecise	Function is precise or imprecise.	Imprecise functions contain operations such as floating point operations.
IsSystemVerified	The precision and determinism properties of the function can be verified by SQL Server.	
SystemDataAccess	Function accesses system data (system catalogs or virtual system tables) in the local instance of SQL Server.	
UserDataAccess	Function accesses user data in the local instance of SQL Server.	Includes user-defined tables and temp tables, but not table variables.

The precision and determinism properties of Transact-SQL functions are determined automatically by SQL Server. The data access and determinism properties of CLR functions can be specified by the user. For more information, see Overview of SQL CLR Routine Custom Attributes.

To display the current values for these properties, use OBJECTPROPERTYEX.

A computed column that invokes a user-defined function can be used in an index when the user-defined function has the following property values:

- **IsDeterministic** = true
- **IsSystemVerified** = true (unless the computed column is persisted)
- **UserDataAccess** = false
- **SystemDataAccess** = false

For more information, see Creating Indexes on Computed Columns.

Calling Extended Stored Procedures from Functions

The extended stored procedure, when it is called from inside a function, cannot return result sets to the client. Any ODS APIs that return result sets to the client will return FAIL. The extended stored procedure could connect back to an instance of SQL Server; however, it should not try to join the same transaction as the function that invoked the extended stored procedure.

Similar to invocations from a batch or stored procedure, the extended stored procedure will be executed in the context of the Windows security account under which SQL Server is running. The

owner of the stored procedure should consider this when giving EXECUTE permission on it to users.

Limitations and Restrictions

User-defined functions cannot be used to perform actions that modify the database state.

User-defined functions cannot contain an OUTPUT INTO clause that has a table as its target.

The following Service Broker statements cannot be included in the definition of a Transact-SQL user-defined function:

- BEGIN DIALOG CONVERSATION
- END CONVERSATION
- GET CONVERSATION GROUP
- MOVE CONVERSATION
- RECEIVE
- SEND

User-defined functions can be nested; that is, one user-defined function can call another. The nesting level is incremented when the called function starts execution, and decremented when the called function finishes execution. User-defined functions can be nested up to 32 levels. Exceeding the maximum levels of nesting causes the whole calling function chain to fail. Any reference to managed code from a Transact-SQL user-defined function counts as one level against the 32-level nesting limit. Methods invoked from within managed code do not count against this limit.

Using Sort Order in CLR Table-valued Functions

When using the ORDER clause in CLR table-valued functions, follow these guidelines:

- You must ensure that results are always ordered in the specified order. If the results are not in the specified order, SQL Server will generate an error message when the query is executed.
- If an ORDER clause is specified, the output of the table-valued function must be sorted according to the collation of the column (explicit or implicit). For example, if the column collation is Chinese (either specified in the DDL for the table-valued function or obtained from the database collation), the returned results must be sorted according to Chinese sorting rules.
- The ORDER clause, if specified, is always verified by SQL Server while returning results, whether or not it is used by the query processor to perform further optimizations. Only use the ORDER clause if you know it is useful to the query processor.
- The SQL Server query processor takes advantage of the ORDER clause automatically in following cases:
  - Insert queries where the ORDER clause is compatible with an index.
  - ORDER BY clauses that are compatible with the ORDER clause.
  - Aggregates, where GROUP BY is compatible with ORDER clause.

- DISTINCT aggregates where the distinct columns are compatible with the ORDER clause.

The ORDER clause does not guarantee ordered results when a SELECT query is executed, unless ORDER BY is also specified in the query. See sys.function_order_columns (Transact-SQL) for information on how to query for columns included in the sort-order for table-valued functions.

Metadata

The following table lists the system catalog views that you can use to return metadata about user-defined functions.

System View	Description

sys.sql_modules	Displays the definition of Transact-SQL user-defined functions. For example:
	`USE AdventureWorks2012;`
	`GO`
	`SELECT definition, type`
	`FROM sys.sql_modules AS m`
	`JOIN sys.objects AS o ON m.object_id = o.object_id`
	`AND type IN ('FN', 'IF', 'TF');`
	`GO`
	The definition of functions created by using the ENCRYPTION option cannot be viewed by using sys.sql_modules; however, other information about the encrypted functions is displayed.

sys.assembly_modules	Displays information about CLR user-defined functions.
sys.parameters	Displays information about the parameters defined in user-defined functions.
sys.sql_expression_dependencies	Displays the underlying objects referenced by a function.

Permissions

Requires CREATE FUNCTION permission in the database and ALTER permission on the schema in which the function is being created. If the function specifies a user-defined type, requires EXECUTE permission on the type.

Examples

A. Using a scalar-valued user-defined function that calculates the ISO week

The following example creates the user-defined function `ISOweek`. This function takes a date argument and calculates the ISO week number. For this function to calculate correctly, `SET DATEFIRST 1` must be invoked before the function is called.

The example also shows using the EXECUTE AS clause to specify the security context in which a stored procedure can be executed. In the example, the option `CALLER` specifies that the procedure will be executed in the context of the user that calls it. The other options that you can specify are SELF, OWNER, and user_name.

Here is the function call. Notice that `DATEFIRST` is set to `1`.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID (N'dbo.ISOweek', N'FN') IS NOT NULL
    DROP FUNCTION dbo.ISOweek;
GO
CREATE FUNCTION dbo.ISOweek (@DATE datetime)
RETURNS int
WITH EXECUTE AS CALLER
AS
BEGIN
     DECLARE @ISOweek int;
     SET @ISOweek= DATEPART(wk,@DATE)+1
          -DATEPART(wk,CAST(DATEPART(yy,@DATE) as CHAR(4))+'0104');
--Special cases: Jan 1-3 may belong to the previous year
     IF (@ISOweek=0)
          SET @ISOweek=dbo.ISOweek(CAST(DATEPART(yy,@DATE)-1
               AS CHAR(4))+'12'+ CAST(24+DATEPART(DAY,@DATE) AS CHAR(2)))+1;
--Special case: Dec 29-31 may belong to the next year
     IF ((DATEPART(mm,@DATE)=12) AND
          ((DATEPART(dd,@DATE)-DATEPART(dw,@DATE))>= 28))
          SET @ISOweek=1;
     RETURN(@ISOweek);
END;
GO
SET DATEFIRST 1;
SELECT dbo.ISOweek(CONVERT(DATETIME,'12/26/2004',101)) AS 'ISO Week';
```

Here is the result set.

```
ISO Week
----------------
52
```

B. Creating an inline table-valued function

The following example returns an inline table-valued function. It returns three columns `ProductID`, `Name` and the aggregate of year-to-date totals by store as `YTD Total` for each product sold to the store.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID (N'Sales.ufn_SalesByStore', N'IF') IS NOT NULL
    DROP FUNCTION Sales.ufn_SalesByStore;
GO
CREATE FUNCTION Sales.ufn_SalesByStore (@storeid int)
RETURNS TABLE
AS
RETURN
(
    SELECT P.ProductID, P.Name, SUM(SD.LineTotal) AS 'Total'
    FROM Production.Product AS P
    JOIN Sales.SalesOrderDetail AS SD ON SD.ProductID = P.ProductID
    JOIN Sales.SalesOrderHeader AS SH ON SH.SalesOrderID = SD.SalesOrderID
    JOIN Sales.Customer AS C ON SH.CustomerID = C.CustomerID
    WHERE C.StoreID = @storeid
    GROUP BY P.ProductID, P.Name
);
GO
```

To invoke the function, run this query.

```
SELECT * FROM Sales.ufn_SalesByStore (602);
```

C. Creating a multi-statement table-valued function

The following example creates the table-valued function `fn_FindReports(InEmpID)`. When supplied with a valid employee ID, the function returns a table that corresponds to all the employees that report to the employee either directly or indirectly. The function uses a recursive common table expression (CTE) to produce the hierarchical list of employees. For more information about recursive CTEs, see WITH common_table_expression (Transact-SQL).

```
USE AdventureWorks2012;
GO
```

```
IF OBJECT_ID (N'dbo.ufn_FindReports', N'TF') IS NOT NULL
    DROP FUNCTION dbo.ufn_FindReports;
GO
CREATE FUNCTION dbo.ufn_FindReports (@InEmpID INTEGER)
RETURNS @retFindReports TABLE
(
    EmployeeID int primary key NOT NULL,
    FirstName nvarchar(255) NOT NULL,
    LastName nvarchar(255) NOT NULL,
    JobTitle nvarchar(50) NOT NULL,
    RecursionLevel int NOT NULL
)
--Returns a result set that lists all the employees who report to the
--specific employee directly or indirectly.*/
AS
BEGIN
WITH EMP_cte(EmployeeID, OrganizationNode, FirstName, LastName, JobTitle,
RecursionLevel) -- CTE name and columns
    AS (
        SELECT e.BusinessEntityID, e.OrganizationNode, p.FirstName,
p.LastName, e.JobTitle, 0 -- Get the initial list of Employees for Manager n
        FROM HumanResources.Employee e
            INNER JOIN Person.Person p
            ON p.BusinessEntityID = e.BusinessEntityID
        WHERE e.BusinessEntityID = @InEmpID
        UNION ALL
        SELECT e.BusinessEntityID, e.OrganizationNode, p.FirstName,
p.LastName, e.JobTitle, RecursionLevel + 1 -- Join recursive member to anchor
        FROM HumanResources.Employee e
            INNER JOIN EMP_cte
            ON e.OrganizationNode.GetAncestor(1) = EMP_cte.OrganizationNode
            INNER JOIN Person.Person p
            ON p.BusinessEntityID = e.BusinessEntityID
        )
-- copy the required columns to the result of the function
   INSERT @retFindReports
```

```
   SELECT EmployeeID, FirstName, LastName, JobTitle, RecursionLevel
   FROM EMP_cte
   RETURN
END;
GO
-- Example invocation
SELECT EmployeeID, FirstName, LastName, JobTitle, RecursionLevel
FROM dbo.ufn_FindReports(1);

GO
```

D. Creating a CLR function

The example creates CLR function `len_s`. Before the function is created, the assembly `SurrogateStringFunction.dll` is registered in the local database.

```
DECLARE @SamplesPath nvarchar(1024);
-- You may have to modify the value of this variable if you have
-- installed the sample in a location other than the default location.
SELECT @SamplesPath = REPLACE(physical_name, 'Microsoft SQL
Server\MSSQL11.MSSQLSERVER\MSSQL\DATA\master.mdf', 'Microsoft SQL
Server\100\Samples\Engine\Programmability\CLR\')
    FROM master.sys.database_files
    WHERE name = 'master';

CREATE ASSEMBLY [SurrogateStringFunction]
FROM @SamplesPath +
'StringManipulate\CS\StringManipulate\bin\debug\SurrogateStringFunction.dll'
WITH PERMISSION_SET = EXTERNAL_ACCESS;
GO

CREATE FUNCTION [dbo].[len_s] (@str nvarchar(4000))
RETURNS bigint
AS EXTERNAL NAME
[SurrogateStringFunction].[Microsoft.Samples.SqlServer.SurrogateStringFunctio
n].[LenS];
GO
```

For an example of how to create a CLR table-valued function, see CLR Table-Valued Functions.

See Also

ALTER FUNCTION
DROP FUNCTION
OBJECTPROPERTYEX
sys.sql_modules (Transact-SQL)
sys.assembly_modules
EXECUTE (Transact-SQL)
CLR User-Defined Functions
EVENTDATA (Transact-SQL)

CREATE INDEX

Creates a relational index on a specified table or view on a specified table. An index can be created before there is data in the table. Relational indexes can be created on tables or views in another database by specifying a qualified database name.

Note

For information about how to create an XML index, see CREATE XML INDEX (Transact-SQL). For information about how to create a spatial index, see CREATE SPATIAL INDEX (Transact-SQL). For information about how to create an xVelocity memory optimized columnstore index, see CREATE COLUMNSTORE INDEX (Transact-SQL).


Transact-SQL Syntax Conventions

Syntax

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index_name
    ON <object> ( column [ ASC | DESC ] [ ,...n ] )
    [ INCLUDE ( column_name [ ,...n ] ) ]
    [ WHERE <filter_predicate> ]
    [ WITH ( <relational_index_option> [ ,...n ] ) ]
    [ ON { partition_scheme_name ( column_name )
         | filegroup_name
         | default
         }
    ]
    [ FILESTREAM_ON { filestream_filegroup_name | partition_scheme_name | "NULL" } ]

[ ; ]
```

<object> ::=

{

[**database_name.** [**schema_name**] **.** | **schema_name.**]

table_or_view_name

}

<relational_index_option> ::=

{

PAD_INDEX = { ON | OFF }

| FILLFACTOR = **fillfactor**

| SORT_IN_TEMPDB = { ON | OFF }

| IGNORE_DUP_KEY = { ON | OFF }

| STATISTICS_NORECOMPUTE = { ON | OFF }

| DROP_EXISTING = { ON | OFF }

| ONLINE = { ON | OFF }

| ALLOW_ROW_LOCKS = { ON | OFF }

| ALLOW_PAGE_LOCKS = { ON | OFF }

| MAXDOP = **max_degree_of_parallelism**

| DATA_COMPRESSION = { NONE | ROW | PAGE}

[ON PARTITIONS **(** { <partition_number_expression> | <range> }

[**,** ...**n**] **)**]

}

<filter_predicate> ::=

<conjunct> [AND <conjunct>]

<conjunct> ::=

<disjunct> | <comparison>

<disjunct> ::=

column_name IN (**constant** ,...**n**)

<comparison> ::=

column_name <comparison_op> **constant**

<comparison_op> ::=

{ IS | IS NOT | = | <> | != | > | >= | !> | < | <= | !< }

<range> ::=

<partition_number_expression> TO <partition_number_expression>

Backward Compatible Relational Index

Important The backward compatible relational index syntax structure will be removed in a future version of SQL Server. Avoid using this syntax structure in new development work, and plan to modify applications that currently use the feature. Use the syntax structure specified in **<relational_index_option>** instead.

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index_name
    ON <object> ( column_name [ ASC | DESC ] [ ,...n ] )
    [ WITH <backward_compatible_index_option> [ ,...n ] ]
    [ ON { filegroup_name | "default" } ]

<object> ::=
{
    [ database_name. [ owner_name ] . | owner_name. ]
    table_or_view_name
}

<backward_compatible_index_option> ::=
{
    PAD_INDEX
  | FILLFACTOR = fillfactor
  | SORT_IN_TEMPDB
  | IGNORE_DUP_KEY
  | STATISTICS_NORECOMPUTE
  | DROP_EXISTING
}
```

Arguments

UNIQUE

Creates a unique index on a table or view. A unique index is one in which no two rows are

permitted to have the same index key value. A clustered index on a view must be unique.

The Database Engine does not allow creating a unique index on columns that already include duplicate values, whether or not IGNORE_DUP_KEY is set to ON. If this is tried, the Database Engine displays an error message. Duplicate values must be removed before a unique index can be created on the column or columns. Columns that are used in a unique index should be set to NOT NULL, because multiple null values are considered duplicates when a unique index is created.

CLUSTERED

Creates an index in which the logical order of the key values determines the physical order of the corresponding rows in a table. The bottom, or leaf, level of the clustered index contains the actual data rows of the table. A table or view is allowed one clustered index at a time.

A view with a unique clustered index is called an indexed view. Creating a unique clustered index on a view physically materializes the view. A unique clustered index must be created on a view before any other indexes can be defined on the same view. For more information, see Create Indexed Views.

Create the clustered index before creating any nonclustered indexes. Existing nonclustered indexes on tables are rebuilt when a clustered index is created.

If CLUSTERED is not specified, a nonclustered index is created.

Note

Because the leaf level of a clustered index and the data pages are the same by definition, creating a clustered index and using the ON partition_scheme_name or ON filegroup_name clause effectively moves a table from the filegroup on which the table was created to the new partition scheme or filegroup. Before creating tables or indexes on specific filegroups, verify which filegroups are available and that they have enough empty space for the index.

In some cases creating a clustered index can enable previously disabled indexes. For more information, see Enable Indexes and Constraints and Disable Indexes and Constraints.

NONCLUSTERED

Creates an index that specifies the logical ordering of a table. With a nonclustered index, the physical order of the data rows is independent of their indexed order.

Each table can have up to 999 nonclustered indexes, regardless of how the indexes are created: either implicitly with PRIMARY KEY and UNIQUE constraints, or explicitly with CREATE INDEX.

For indexed views, nonclustered indexes can be created only on a view that has a unique clustered index already defined.

The default is NONCLUSTERED.

index_name

Is the name of the index. Index names must be unique within a table or view but do not have

to be unique within a database. Index names must follow the rules of identifiers.

column

Is the column or columns on which the index is based. Specify two or more column names to create a composite index on the combined values in the specified columns. List the columns to be included in the composite index, in sort-priority order, inside the parentheses after table_or_view_name.

Up to 16 columns can be combined into a single composite index key. All the columns in a composite index key must be in the same table or view. The maximum allowable size of the combined index values is 900 bytes.

Columns that are of the large object (LOB) data types **ntext**, **text**, **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, **xml**, or **image** cannot be specified as key columns for an index. Also, a view definition cannot include **ntext**, **text**, or **image** columns, even if they are not referenced in the CREATE INDEX statement.

You can create indexes on CLR user-defined type columns if the type supports binary ordering. You can also create indexes on computed columns that are defined as method invocations off a user-defined type column, as long as the methods are marked deterministic and do not perform data access operations. For more information about indexing CLR user-defined type columns, see CLR User-defined Types.

[ASC | DESC]

Determines the ascending or descending sort direction for the particular index column. The default is ASC.

INCLUDE (column [,... n])

Specifies the non-key columns to be added to the leaf level of the nonclustered index. The nonclustered index can be unique or non-unique.

Column names cannot be repeated in the INCLUDE list and cannot be used simultaneously as both key and non-key columns. Nonclustered indexes always contain the clustered index columns if a clustered index is defined on the table. For more information, see Index with Included Columns.

All data types are allowed except **text**, **ntext**, and **image**. The index must be created or rebuilt offline (ONLINE = OFF) if any one of the specified non-key columns are **varchar(max)**, **nvarchar(max)**, or **varbinary(max)** data types.

Computed columns that are deterministic and either precise or imprecise can be included columns. Computed columns derived from **image**, **ntext**, **text**, **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, and **xml** data types can be included in non-key columns as long as the computed column data types is allowable as an included column. For more information, see Creating Indexes on Computed Columns.

For information on creating an XML index, see CREATE XML INDEX (Transact-SQL).

WHERE <filter_predicate>

Creates a filtered index by specifying which rows to include in the index. The filtered index

must be a nonclustered index on a table. Creates filtered statistics for the data rows in the filtered index.

The filter predicate uses simple comparison logic and cannot reference a computed column, a UDT column, a spatial data type column, or a hierarchyID data type column. Comparisons using NULL literals are not allowed with the comparison operators. Use the IS NULL and IS NOT NULL operators instead.

Here are some examples of filter predicates for the `Production.BillOfMaterials` table:

```
WHERE StartDate > '20000101' AND EndDate <= '20000630'
```

```
WHERE ComponentID IN (533, 324, 753)
```

```
WHERE StartDate IN ('20000404', '20000905') AND EndDate IS
NOT NULL
```

Filtered indexes do not apply to XML indexes and full-text indexes. For UNIQUE indexes, only the selected rows must have unique index values. Filtered indexes do not allow the IGNORE_DUP_KEY option.

ON partition_scheme_name (column_name)

Specifies the partition scheme that defines the filegroups onto which the partitions of a partitioned index will be mapped. The partition scheme must exist within the database by executing either CREATE PARTITION SCHEME or ALTER PARTITION SCHEME. column_name specifies the column against which a partitioned index will be partitioned. This column must match the data type, length, and precision of the argument of the partition function that partition_scheme_name is using. column_name is not restricted to the columns in the index definition. Any column in the base table can be specified, except when partitioning a UNIQUE index, column_name must be chosen from among those used as the unique key. This restriction allows the Database Engine to verify uniqueness of key values within a single partition only.

Note

When you partition a non-unique, clustered index, the Database Engine by default adds the partitioning column to the list of clustered index keys, if it is not already specified. When partitioning a non-unique, nonclustered index, the Database Engine adds the partitioning column as a non-key (included) column of the index, if it is not already specified.

If partition_scheme_name or filegroup is not specified and the table is partitioned, the index is placed in the same partition scheme, using the same partitioning column, as the underlying table.

Note

You cannot specify a partitioning scheme on an XML index. If the base table is partitioned, the XML index uses the same partition scheme as the table.

For more information about partitioning indexes, Partitioned Tables and Indexes.

ON filegroup_name

Creates the specified index on the specified filegroup. If no location is specified and the table or view is not partitioned, the index uses the same filegroup as the underlying table or view. The filegroup must already exist.

ON "default"

Creates the specified index on the default filegroup.

The term default, in this context, is not a keyword. It is an identifier for the default filegroup and must be delimited, as in ON **"**default**"** or ON **[**default**]**. If "default" is specified, the QUOTED_IDENTIFIER option must be ON for the current session. This is the default setting. For more information, see SET QUOTED_IDENTIFIER (Transact-SQL).

[FILESTREAM_ON { filestream_filegroup_name | partition_scheme_name | "NULL" }]

Specifies the placement of FILESTREAM data for the table when a clustered index is created. The FILESTREAM_ON clause allows FILESTREAM data to be moved to a different FILESTREAM filegroup or partition scheme.

filestream_filegroup_name is the name of a FILESTREAM filegroup. The filegroup must have one file defined for the filegroup by using a CREATE DATABASE or ALTER DATABASE statement; otherwise, an error is raised.

If the table is partitioned, the FILESTREAM_ON clause must be included and must specify a partition scheme of FILESTREAM filegroups that uses the same partition function and partition columns as the partition scheme for the table. Otherwise, an error is raised.

If the table is not partitioned, the FILESTREAM column cannot be partitioned. FILESTREAM data for the table must be stored in a single filegroup that is specified in the FILESTREAM_ON clause.

FILESTREAM_ON NULL can be specified in a CREATE INDEX statement if a clustered index is being created and the table does not contain a FILESTREAM column.

For more information, see FILESTREAM (SQL Server).

<object>::=

Is the fully qualified or nonfully qualified object to be indexed.

database_name

Is the name of the database.

schema_name

Is the name of the schema to which the table or view belongs.

table_or_view_name

Is the name of the table or view to be indexed.

The view must be defined with SCHEMABINDING to create an index on it. A unique clustered index must be created on a view before any nonclustered index is created. For more information about indexed views, see the Remarks section.

<relational_index_option>::=

Specifies the options to use when you create the index.

PAD_INDEX = { ON | OFF }

Specifies index padding. The default is OFF.

ON

The percentage of free space that is specified by fillfactor is applied to the intermediate-level pages of the index.

OFF or fillfactor is not specified

The intermediate-level pages are filled to near capacity, leaving sufficient space for at least one row of the maximum size the index can have, considering the set of keys on the intermediate pages.

The PAD_INDEX option is useful only when FILLFACTOR is specified, because PAD_INDEX uses the percentage specified by FILLFACTOR. If the percentage specified for FILLFACTOR is not large enough to allow for one row, the Database Engine internally overrides the percentage to allow for the minimum. The number of rows on an intermediate index page is never less than two, regardless of how low the value of fillfactor.

In backward compatible syntax, WITH PAD_INDEX is equivalent to WITH PAD_INDEX = ON.

FILLFACTOR = fillfactor

Specifies a percentage that indicates how full the Database Engine should make the leaf level of each index page during index creation or rebuild. fillfactor must be an integer value from 1 to 100. If fillfactor is 100, the Database Engine creates indexes with leaf pages filled to capacity.

The FILLFACTOR setting applies only when the index is created or rebuilt. The Database Engine does not dynamically keep the specified percentage of empty space in the pages. To view the fill factor setting, use the sys.indexes catalog view.

Important

Creating a clustered index with a FILLFACTOR less than 100 affects the amount of storage space the data occupies because the Database Engine redistributes the data when it creates the clustered index.

For more information, see Fill Factor.

SORT_IN_TEMPDB = { ON | OFF }

Specifies whether to store temporary sort results in **tempdb**. The default is OFF.

ON

The intermediate sort results that are used to build the index are stored in **tempdb**. This may reduce the time required to create an index if **tempdb** is on a different set of disks than the user database. However, this increases the amount of disk space that is used during the index build.

OFF

The intermediate sort results are stored in the same database as the index.

In addition to the space required in the user database to create the index, **tempdb** must have about the same amount of additional space to hold the intermediate sort results. For more information, see tempdb and Index Creation.

In backward compatible syntax, WITH SORT_IN_TEMPDB is equivalent to WITH SORT_IN_TEMPDB = ON.

IGNORE_DUP_KEY = { ON | OFF }

Specifies the error response when an insert operation attempts to insert duplicate key values into a unique index. The IGNORE_DUP_KEY option applies only to insert operations after the index is created or rebuilt. The option has no effect when executing CREATE INDEX, ALTER INDEX, or UPDATE. The default is OFF.

ON

A warning message will occur when duplicate key values are inserted into a unique index. Only the rows violating the uniqueness constraint will fail.

OFF

An error message will occur when duplicate key values are inserted into a unique index. The entire INSERT operation will be rolled back.

IGNORE_DUP_KEY cannot be set to ON for indexes created on a view, non-unique indexes, XML indexes, spatial indexes, and filtered indexes.

To view IGNORE_DUP_KEY, use sys.indexes.

In backward compatible syntax, WITH IGNORE_DUP_KEY is equivalent to WITH IGNORE_DUP_KEY = ON.

STATISTICS_NORECOMPUTE = { ON | OFF }

Specifies whether distribution statistics are recomputed. The default is OFF.

ON

Out-of-date statistics are not automatically recomputed.

OFF

Automatic statistics updating are enabled.

To restore automatic statistics updating, set the STATISTICS_NORECOMPUTE to OFF, or execute UPDATE STATISTICS without the NORECOMPUTE clause.

Important

Disabling automatic recomputation of distribution statistics may prevent the query optimizer from picking optimal execution plans for queries involving the table.

In backward compatible syntax, WITH STATISTICS_NORECOMPUTE is equivalent to WITH STATISTICS_NORECOMPUTE = ON.

DROP_EXISTING = { ON | OFF }

Specifies that the named, preexisting clustered, or nonclustered is dropped and rebuilt. The default is OFF.

ON

The existing index is dropped and rebuilt. The index name specified must be the same as a currently existing index; however, the index definition can be modified. For example, you can specify different columns, sort order, partition scheme, or index options.

OFF

An error is displayed if the specified index name already exists.

The index type cannot be changed by using DROP_EXISTING.

In backward compatible syntax, WITH DROP_EXISTING is equivalent to WITH DROP_EXISTING = ON.

ONLINE = { ON | OFF }

Specifies whether underlying tables and associated indexes are available for queries and data modification during the index operation. The default is OFF.

Note

Online index operations are not available in every edition of Microsoft SQL Server. For a list of features that are supported by the editions of SQL Server, see Features Supported by the Editions of SQL Server 2012.

ON

Long-term table locks are not held for the duration of the index operation. During the main phase of the index operation, only an Intent Share (IS) lock is held on the source table. This enables queries or updates to the underlying table and indexes to proceed. At the start of the operation, a Shared (S) lock is held on the source object for a very short period of time. At the end of the operation, for a short period of time, an S (Shared) lock is acquired on the source if a nonclustered index is being created; or an SCH-M (Schema Modification) lock is acquired when a clustered index is created or dropped online and when a clustered or nonclustered index is being rebuilt. ONLINE cannot be set to ON when an index is being created on a local temporary table.

OFF

Table locks are applied for the duration of the index operation. An offline index operation that creates, rebuilds, or drops a clustered index, or rebuilds or drops a nonclustered index, acquires a Schema modification (Sch-M) lock on the table. This prevents all user access to the underlying table for the duration of the operation. An offline index operation that creates a nonclustered index acquires a Shared (S) lock on the table. This prevents updates to the underlying table but allows read operations, such as SELECT statements.

For more information, see How Online Index Operations Work.

Indexes, including indexes on global temp tables, can be created online with the following

exceptions:

- XML index
- Index on a local temp table.
- Initial unique clustered index on a view.
- Disabled clustered indexes.
- Clustered index if the underlying table contains LOB data types: **image**, **ntext**, **text**, and spatial types.

For more information, see Performing Index Operations Online.

ALLOW_ROW_LOCKS = { ON | OFF }

Specifies whether row locks are allowed. The default is ON.

ON

Row locks are allowed when accessing the index. The Database Engine determines when row locks are used.

OFF

Row locks are not used.

ALLOW_PAGE_LOCKS = { ON | OFF }

Specifies whether page locks are allowed. The default is ON.

ON

Page locks are allowed when accessing the index. The Database Engine determines when page locks are used.

OFF

Page locks are not used.

MAXDOP = max_degree_of_parallelism

Overrides the Configure the max degree of parallelism Server Configuration Option configuration option for the duration of the index operation. Use MAXDOP to limit the number of processors used in a parallel plan execution. The maximum is 64 processors.

max_degree_of_parallelism can be:

1

Suppresses parallel plan generation.

>1

Restricts the maximum number of processors used in a parallel index operation to the specified number or fewer based on the current system workload.

0 (default)

Uses the actual number of processors or fewer based on the current system workload.

For more information, see Configuring Parallel Index Operations.

Note

Parallel index operations are not available in every edition of Microsoft SQL Server. For a list of features that are supported by the editions of SQL Server, see Features Supported by the Editions of SQL Server 2012.

DATA_COMPRESSION

Specifies the data compression option for the specified index, partition number, or range of partitions. The options are as follows:

NONE

Index or specified partitions are not compressed.

ROW

Index or specified partitions are compressed by using row compression.

PAGE

Index or specified partitions are compressed by using page compression.

For more information about compression, see Creating Compressed Tables and Indexes.

ON PARTITIONS ({ <partition_number_expression> | <range> } [,...n])

Specifies the partitions to which the DATA_COMPRESSION setting applies. If the index is not partitioned, the ON PARTITIONS argument will generate an error. If the ON PARTITIONS clause is not provided, the DATA_COMPRESSION option applies to all partitions of a partitioned index.

<partition_number_expression> can be specified in the following ways:

- Provide the number for a partition, for example: ON PARTITIONS (2).
- Provide the partition numbers for several individual partitions separated by commas, for example: ON PARTITIONS (1, 5).
- Provide both ranges and individual partitions, for example: ON PARTITIONS (2, 4, 6 TO 8).

<range> can be specified as partition numbers separated by the word TO, for example: ON PARTITIONS (6 TO 8).

To set different types of data compression for different partitions, specify the DATA_COMPRESSION option more than once, for example:

```
REBUILD WITH
(
DATA_COMPRESSION = NONE ON PARTITIONS (1),
DATA_COMPRESSION = ROW ON PARTITIONS (2, 4, 6 TO 8),
DATA_COMPRESSION = PAGE ON PARTITIONS (3, 5)
)
```

Remarks

The CREATE INDEX statement is optimized like any other query. To save on I/O operations, the query processor may choose to scan another index instead of performing a table scan. The sort operation may be eliminated in some situations. On multiprocessor computers CREATE INDEX can use more processors to perform the scan and sort operations associated with creating the index, in the same way as other queries do. For more information, see Configuring Parallel Index Operations.

The create index operation can be minimally logged if the database recovery model is set to either bulk-logged or simple.

Indexes can be created on a temporary table. When the table is dropped or the session ends, the indexes are dropped.

Indexes support extended properties.

Clustered Indexes

Creating a clustered index on a table (heap) or dropping and re-creating an existing clustered index requires additional workspace to be available in the database to accommodate data sorting and a temporary copy of the original table or existing clustered index data. For more information about clustered indexes, see Creating Clustered Indexes.

Unique Indexes

When a unique index exists, the Database Engine checks for duplicate values each time data is added by a insert operations. Insert operations that would generate duplicate key values are rolled back, and the Database Engine displays an error message. This is true even if the insert operation changes many rows but causes only one duplicate. If an attempt is made to enter data for which there is a unique index and the IGNORE_DUP_KEY clause is set to ON, only the rows violating the UNIQUE index fail.

Partitioned Indexes

Partitioned indexes are created and maintained in a similar manner to partitioned tables, but like ordinary indexes, they are handled as separate database objects. You can have a partitioned index on a table that is not partitioned, and you can have a nonpartitioned index on a table that is partitioned.

If you are creating an index on a partitioned table, and do not specify a filegroup on which to place the index, the index is partitioned in the same manner as the underlying table. This is because indexes, by default, are placed on the same filegroups as their underlying tables, and for a partitioned table in the same partition scheme that uses the same partitioning columns. When the index uses the same partition scheme and partitioning column as the table, the index is *aligned* with the table.

Warning

Creating and rebuilding nonaligned indexes on a table with more than 1,000 partitions is possible, but is not supported. Doing so may cause degraded performance or excessive

memory consumption during these operations. We recommend using only aligned indexes when the number of partitions exceed 1,000.

When partitioning a non-unique, clustered index, the Database Engine by default adds any partitioning columns to the list of clustered index keys, if not already specified.

Indexed views can be created on partitioned tables in the same manner as indexes on tables. For more information about partitioned indexes, see Partitioned Tables and Indexes.

In SQL Server 2012, statistics are not created by scanning all the rows in the table when a partitioned index is created or rebuilt. Instead, the query optimizer uses the default sampling algorithm to generate statistics. To obtain statistics on partitioned indexes by scanning all the rows in the table, use CREATE STATISTICS or UPDATE STATISTICS with the FULLSCAN clause.

Filtered Indexes

A filtered index is an optimized nonclustered index, suited for queries that select a small percentage of rows from a table. It uses a filter predicate to index a portion of the data in the table. A well-designed filtered index can improve query performance, reduce storage costs, and reduce maintenance costs.

Required SET Options for Filtered Indexes

The SET options in the Required Value column are required whenever any of the following conditions occur:

- Create a filtered index.
- INSERT, UPDATE, DELETE, or MERGE operation modifies the data in a filtered index.
- The query optimizer uses the filtered index in the query execution plan.

SET options	Required value
ANSI_NULLS	ON
ANSI_PADDING	ON
ANSI_WARNINGS*	ON
ARITHABORT	ON
CONCAT_NULL_YIELDS_NULL	ON
NUMERIC_ROUNDABORT	OFF
QUOTED_IDENTIFIER	ON

*Setting ANSI_WARNINGS to ON implicitly sets ARITHABORT to ON when the database compatibility level is set to 90 or higher. If the database compatibility level is set to 80 or earlier, the ARITHABORT option must explicitly be set to ON.

If the SET options are incorrect, the following conditions can occur:

- The filtered index is not created.
- The Database Engine generates an error and rolls back INSERT, UPDATE, DELETE, or MERGE statements that change data in the index.
- Query optimizer does not consider the index in the execution plan for any Transact-SQL statements.

For more information about Filtered Indexes, see Filtered Index Design Guidelines.

Spatial Indexes

For information about spatial indexes, see CREATE SPATIAL INDEX (Transact-SQL) and Working with Spatial Indexes.

XML Indexes

For information about XML indexes see, CREATE XML INDEX (Transact-SQL) and Indexes on xml Type columns.

Index Key Size

The maximum size for an index key is 900 bytes. Indexes on **varchar** columns that exceed 900 bytes can be created if the existing data in the columns do not exceed 900 bytes at the time the index is created; however, subsequent insert or update actions on the columns that cause the total size to be greater than 900 bytes will fail. The index key of a clustered index cannot contain **varchar** columns that have existing data in the ROW_OVERFLOW_DATA allocation unit. If a clustered index is created on a **varchar** column and the existing data is in the IN_ROW_DATA allocation unit, subsequent insert or update actions on the column that would push the data off-row will fail.

Nonclustered indexes can include non-key columns in the leaf level of the index. These columns are not considered by the Database Engine when calculating the index key size . For more information, see Index with Included Columns.

Note
When tables are partitioned, if the partitioning key columns are not already present in a non-unique clustered index, they are added to the index by the Database Engine. The combined size of the indexed columns (not counting included columns), plus any added partitioning columns cannot exceed 1800 bytes in a non-unique clustered index.

Computed Columns

Indexes can be created on computed columns. In addition, computed columns can have the property PERSISTED. This means that the Database Engine stores the computed values in the table, and updates them when any other columns on which the computed column depends are updated. The Database Engine uses these persisted values when it creates an index on the column, and when the index is referenced in a query.

To index a computed column, the computed column must deterministic and precise. However, using the PERSISTED property expands the type of indexable computed columns to include:

- Computed columns based on Transact-SQL and CLR functions and CLR user-defined type methods that are marked deterministic by the user.
- Computed columns based on expressions that are deterministic as defined by the Database Engine but imprecise.

Persisted computed columns require the following SET options to be set as shown in the previous section "Required SET Options for Indexed Views".

The UNIQUE or PRIMARY KEY constraint can contain a computed column as long as it satisfies all conditions for indexing. Specifically, the computed column must be deterministic and precise or deterministic and persisted. For more information about determinism, see Deterministic and Nondeterministic Functions.

Computed columns derived from **image**, **ntext**, **text**, **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, and **xml** data types can be indexed either as a key or included non-key column as long as the computed column data type is allowable as an index key column or non-key column. For example, you cannot create a primary XML index on a computed **xml** column. If the index key size exceeds 900 bytes, a warning message is displayed.

Creating an index on a computed column may cause the failure of an insert or update operation that previously worked. Such a failure may take place when the computed column results in arithmetic error. For example, in the following table, although computed column `c` results in an arithmetic error, the `INSERT` statement works.

```
CREATE TABLE t1 (a int, b int, c AS a/b);

INSERT INTO t1 VALUES (1, 0);
```

If, instead, after creating the table, you create an index on computed column `c`, the same `INSERT` statement will now fail.

```
CREATE TABLE t1 (a int, b int, c AS a/b);

CREATE UNIQUE CLUSTERED INDEX Idx1 ON t1(c);

INSERT INTO t1 VALUES (1, 0);
```

For more information, see Creating Indexes on Computed Columns.

Included Columns in Indexes

Non-key columns, called included columns, can be added to the leaf level of a nonclustered index to improve query performance by covering the query. That is, all columns referenced in the query are included in the index as either key or non-key columns. This allows the query optimizer to locate all the required information from an index scan; the table or clustered index data is not accessed. For more information, see Index with Included Columns.

Specifying Index Options

SQL Server 2005 introduced new index options and also modifies the way in which options are specified. In backward compatible syntax, WITH option_name is equivalent to WITH **(** <option_name> = ON **)**. When you set index options, the following rules apply:

- New index options can only be specified by using WITH **(**option_name = ON | OFF**)**.

- Options cannot be specified by using both the backward compatible and new syntax in the same statement. For example, specifying WITH **(**DROP_EXISTING**,** ONLINE **=** ON**)** causes the statement to fail.
- When you create an XML index, the options must be specified by using WITH **(**option_name **=** ON | OFF**)**.

DROP_EXISTING Clause

You can use the DROP_EXISTING clause to rebuild the index, add or drop columns, modify options, modify column sort order, or change the partition scheme or filegroup.

If the index enforces a PRIMARY KEY or UNIQUE constraint and the index definition is not altered in any way, the index is dropped and re-created preserving the existing constraint. However, if the index definition is altered the statement fails. To change the definition of a PRIMARY KEY or UNIQUE constraint, drop the constraint and add a constraint with the new definition.

DROP_EXISTING enhances performance when you re-create a clustered index, with either the same or different set of keys, on a table that also has nonclustered indexes. DROP_EXISTING replaces the execution of a DROP INDEX statement on the old clustered index followed by the execution of a CREATE INDEX statement for the new clustered index. The nonclustered indexes are rebuilt once, and then only if the index definition has changed. The DROP_EXISTING clause does not rebuild the nonclustered indexes when the index definition has the same index name, key and partition columns, uniqueness attribute, and sort order as the original index.

Whether the nonclustered indexes are rebuilt or not, they always remain in their original filegroups or partition schemes and use the original partition functions. If a clustered index is rebuilt to a different filegroup or partition scheme, the nonclustered indexes are not moved to coincide with the new location of the clustered index. Therefore, even the nonclustered indexes previously aligned with the clustered index, they may no longer be aligned with it. For more information about partitioned index alignment, see.

The DROP_EXISTING clause will not sort the data again if the same index key columns are used in the same order and with the same ascending or descending order, unless the index statement specifies a nonclustered index and the ONLINE option is set to OFF. If the clustered index is disabled, the CREATE INDEX WITH DROP_EXISTING operation must be performed with ONLINE set to OFF. If a nonclustered index is disabled and is not associated with a disabled clustered index, the CREATE INDEX WITH DROP_EXISTING operation can be performed with ONLINE set to OFF or ON.

When indexes with 128 extents or more are dropped or rebuilt, the Database Engine defers the actual page deallocations, and their associated locks, until after the transaction commits.

ONLINE Option

The following guidelines apply for performing index operations online:

- The underlying table cannot be altered, truncated, or dropped while an online index operation is in process.
- Additional temporary disk space is required during the index operation.

- Online operations can be performed on partitioned indexes and indexes that contain persisted computed columns, or included columns.

For more information, see Performing Index Operations Online.

Row and Page Locks Options

When ALLOW_ROW_LOCKS = ON and ALLOW_PAGE_LOCK = ON, row-, page-, and table-level locks are allowed when accessing the index. The Database Engine chooses the appropriate lock and can escalate the lock from a row or page lock to a table lock.

When ALLOW_ROW_LOCKS = OFF and ALLOW_PAGE_LOCK = OFF, only a table-level lock is allowed when accessing the index.

Viewing Index Information

To return information about indexes, you can use catalog views, system functions, and system stored procedures.

Data Compression

Data compression is described in the topic Creating Compressed Tables and Indexes. The following are key points to consider:

- Compression can allow more rows to be stored on a page, but does not change the maximum row size.
- Non-leaf pages of an index are not page compressed but can be row compressed.
- Each nonclustered index has an individual compression setting, and does not inherit the compression setting of the underlying table.
- When a clustered index is created on a heap, the clustered index inherits the compression state of the heap unless an alternative compression state is specified.

The following restrictions apply to partitioned indexes:

- You cannot change the compression setting of a single partition if the table has nonaligned indexes.
- The ALTER INDEX <index> ... REBUILD PARTITION ... syntax rebuilds the specified partition of the index.
- The ALTER INDEX <index> ... REBUILD WITH ... syntax rebuilds all partitions of the index.

To evaluate how changing the compression state will affect a table, an index, or a partition, use the sp_estimate_data_compression_savings stored procedure.

Permissions

Requires ALTER permission on the table or view. User must be a member of the **sysadmin** fixed server role or the **db_ddladmin** and **db_owner** fixed database roles.

Examples

A. Creating a simple nonclustered index

The following example creates a nonclustered index on the `VendorID` column of the `Purchasing.ProductVendor` table.

```
USE AdventureWorks2012;
GO
IF EXISTS (SELECT name FROM sys.indexes
            WHERE name = N'IX_ProductVendor_VendorID')
    DROP INDEX IX_ProductVendor_VendorID ON Purchasing.ProductVendor;
GO
CREATE INDEX IX_ProductVendor_VendorID
    ON Purchasing.ProductVendor (BusinessEntityID);
GO
```

B. Creating a simple nonclustered composite index

The following example creates a nonclustered composite index on the `SalesQuota` and `SalesYTD` columns of the `Sales.SalesPerson` table.

```
USE AdventureWorks2012
GO
IF EXISTS (SELECT name FROM sys.indexes
            WHERE name = N'IX_SalesPerson_SalesQuota_SalesYTD')
    DROP INDEX IX_SalesPerson_SalesQuota_SalesYTD ON Sales.SalesPerson ;
GO
CREATE NONCLUSTERED INDEX IX_SalesPerson_SalesQuota_SalesYTD
    ON Sales.SalesPerson (SalesQuota, SalesYTD);
GO
```

C. Creating a unique nonclustered index

The following example creates a unique nonclustered index on the `Name` column of the `Production.UnitMeasure` table. The index will enforce uniqueness on the data inserted into the `Name` column.

```
USE AdventureWorks2012;
GO
IF EXISTS (SELECT name from sys.indexes
             WHERE name = N'AK_UnitMeasure_Name')
    DROP INDEX AK_UnitMeasure_Name ON Production.UnitMeasure;
GO
CREATE UNIQUE INDEX AK_UnitMeasure_Name
    ON Production.UnitMeasure(Name);
```

```
GO
```

The following query tests the uniqueness constraint by attempting to insert a row with the same value as that in an existing row.

```
--Verify the existing value.
SELECT Name FROM Production.UnitMeasure WHERE Name = N'Ounces';
GO
INSERT INTO Production.UnitMeasure (UnitMeasureCode, Name, ModifiedDate)
    VALUES ('OC', 'Ounces', GetDate());
```

The resulting error message is:

```
Server: Msg 2601, Level 14, State 1, Line 1
Cannot insert duplicate key row in object 'UnitMeasure' with unique index
'AK_UnitMeasure_Name'. The statement has been terminated.
```

D. Using the IGNORE_DUP_KEY option

The following example demonstrates the effect of the `IGNORE_DUP_KEY` option by inserting multiple rows into a temporary table first with the option set to `ON` and again with the option set to `OFF`. A single row is inserted into the `#Test` table that will intentionally cause a duplicate value when the second multiple-row `INSERT` statement is executed. A count of rows in the table returns the number of rows inserted.

```
USE AdventureWorks2012;
GO
CREATE TABLE #Test (C1 nvarchar(10), C2 nvarchar(50), C3 datetime);
GO
CREATE UNIQUE INDEX AK_Index ON #Test (C2)
    WITH (IGNORE_DUP_KEY = ON);
GO
INSERT INTO #Test VALUES (N'OC', N'Ounces', GETDATE());
INSERT INTO #Test SELECT * FROM Production.UnitMeasure;
GO
SELECT COUNT(*)AS [Number of rows] FROM #Test;
GO
DROP TABLE #Test;
GO
```

Here are the results of the second `INSERT` statement.

```
Server: Msg 3604, Level 16, State 1, Line 5 Duplicate key was ignored.
```

```
Number of rows
--------------
38
```

Notice that the rows inserted from the `Production.UnitMeasure` table that did not violate the uniqueness constraint were successfully inserted. A warning was issued and the duplicate row ignored, but the entire transaction was not rolled back.

The same statements are executed again, but with `IGNORE_DUP_KEY` set to `OFF`.

```
USE AdentureWorks2012;
GO
CREATE TABLE #Test (C1 nvarchar(10), C2 nvarchar(50), C3 datetime);
GO
CREATE UNIQUE INDEX AK_Index ON #Test (C2)
    WITH (IGNORE_DUP_KEY = OFF);
GO
INSERT INTO #Test VALUES (N'OC', N'Ounces', GETDATE());
INSERT INTO #Test SELECT * FROM Production.UnitMeasure;
GO
SELECT COUNT(*)AS [Number of rows] FROM #Test;
GO
DROP TABLE #Test;
GO
```

Here are the results of the second `INSERT` statement.

```
Server: Msg 2601, Level 14, State 1, Line 5
Cannot insert duplicate key row in object '#Test' with unique index
'AK_Index'. The statement has been terminated.

Number of rows
--------------
1
```

Notice that none of the rows from the `Production.UnitMeasure` table were inserted into the table even though only one row in the table violated the `UNIQUE` index constraint.

E. Using DROP_EXISTING to drop and re-create an index

The following example drops and re-creates an existing index on the `ProductID` column of the `Production.WorkOrder` table by using the `DROP_EXISTING` option. The options `FILLFACTOR` and `PAD_INDEX` are also set.

```
USE AdventureWorks2012;
GO
CREATE NONCLUSTERED INDEX IX_WorkOrder_ProductID
    ON Production.WorkOrder(ProductID)
    WITH (FILLFACTOR = 80,
        PAD_INDEX = ON,
        DROP_EXISTING = ON);
GO
```

F. Creating an index on a view

The following example creates a view and an index on that view. Two queries are included that use the indexed view.

```
USE AdventureWorks2012;
GO
--Set the options to support indexed views.
SET NUMERIC_ROUNDABORT OFF;
SET ANSI_PADDING, ANSI_WARNINGS, CONCAT_NULL_YIELDS_NULL, ARITHABORT,
    QUOTED_IDENTIFIER, ANSI_NULLS ON;
GO
--Create view with schemabinding.
IF OBJECT_ID ('Sales.vOrders', 'view') IS NOT NULL
DROP VIEW Sales.vOrders ;
GO
CREATE VIEW Sales.vOrders
WITH SCHEMABINDING
AS
    SELECT SUM(UnitPrice*OrderQty*(1.00-UnitPriceDiscount)) AS Revenue,
        OrderDate, ProductID, COUNT_BIG(*) AS COUNT
    FROM Sales.SalesOrderDetail AS od, Sales.SalesOrderHeader AS o
    WHERE od.SalesOrderID = o.SalesOrderID
    GROUP BY OrderDate, ProductID;
GO
--Create an index on the view.
CREATE UNIQUE CLUSTERED INDEX IDX_V1
    ON Sales.vOrders (OrderDate, ProductID);
GO
```

```
--This query can use the indexed view even though the view is
--not specified in the FROM clause.
SELECT SUM(UnitPrice*OrderQty*(1.00-UnitPriceDiscount)) AS Rev,
    OrderDate, ProductID
FROM Sales.SalesOrderDetail AS od
    JOIN Sales.SalesOrderHeader AS o ON od.SalesOrderID=o.SalesOrderID
        AND ProductID BETWEEN 700 and 800
        AND OrderDate >= CONVERT(datetime,'05/01/2002',101)
GROUP BY OrderDate, ProductID
ORDER BY Rev DESC;
GO
--This query can use the above indexed view.
SELECT  OrderDate, SUM(UnitPrice*OrderQty*(1.00-UnitPriceDiscount)) AS Rev
FROM Sales.SalesOrderDetail AS od
    JOIN Sales.SalesOrderHeader AS o ON od.SalesOrderID=o.SalesOrderID
        AND DATEPART(mm,OrderDate)= 3
        AND DATEPART(yy,OrderDate) = 2002
GROUP BY OrderDate
ORDER BY OrderDate ASC;
GO
```

G. Creating an index with included (non-key) columns

The following example creates a nonclustered index with one key column (`PostalCode`) and four non-key columns (`AddressLine1`, `AddressLine2`, `City`, `StateProvinceID`). A query that is covered by the index follows. To display the index that is selected by the query optimizer, on the **Query** menu in SQL Server Management Studio, select **Display Actual Execution Plan** before executing the query.

```
USE AdventureWorks2012;
GO
IF EXISTS (SELECT name FROM sys.indexes
            WHERE name = N'IX_Address_PostalCode')
    DROP INDEX IX_Address_PostalCode ON Person.Address;
GO
CREATE NONCLUSTERED INDEX IX_Address_PostalCode
    ON Person.Address (PostalCode)
    INCLUDE (AddressLine1, AddressLine2, City, StateProvinceID);
GO
```

```
SELECT AddressLine1, AddressLine2, City, StateProvinceID, PostalCode
FROM Person.Address
WHERE PostalCode BETWEEN N'98000' and N'99999';
GO
```

H. Creating a partitioned index

The following example creates a nonclustered partitioned index on `TransactionsPS1`, an existing partition scheme. This example assumes the partitioned index sample has been installed.

```
USE AdentureWorks2012;
GO
IF EXISTS (SELECT name FROM sys.indexes
    WHERE name = N'IX_TransactionHistory_ReferenceOrderID'
    AND object_id = OBJECT_ID(N'Production.TransactionHistory'))
DROP INDEX IX_TransactionHistory_ReferenceOrderID
    ON Production.TransactionHistory;
GO
CREATE NONCLUSTERED INDEX IX_TransactionHistory_ReferenceOrderID
    ON Production.TransactionHistory (ReferenceOrderID)
    ON TransactionsPS1 (TransactionDate);
GO
```

I. Creating a filtered index

The following example creates a filtered index on the Production.BillOfMaterials table. The filter predicate can include columns that are not key columns in the filtered index. The predicate in this example selects only the rows where EndDate is non-NULL.

```
USE AdventureWorks2012;
GO
IF EXISTS (SELECT name FROM sys.indexes
    WHERE name = N'FIBillOfMaterialsWithEndDate'
    AND object_id = OBJECT_ID(N'Production.BillOfMaterials'))
DROP INDEX FIBillOfMaterialsWithEndDate
    ON Production.BillOfMaterials;
GO
CREATE NONCLUSTERED INDEX "FIBillOfMaterialsWithEndDate"
    ON Production.BillOfMaterials (ComponentID, StartDate)
    WHERE EndDate IS NOT NULL;
```

```
GO
```

J. Creating a compressed index

The following example creates an index on a nonpartitioned table by using row compression.

```
CREATE NONCLUSTERED INDEX IX_INDEX_1
    ON T1 (C2)
WITH ( DATA_COMPRESSION = ROW ) ;
GO
```

The following example creates an index on a partitioned table by using row compression on all partitions of the index.

```
CREATE CLUSTERED INDEX IX_PartTab2Col1
ON PartitionTable1 (Col1)
WITH ( DATA_COMPRESSION = ROW ) ;
GO
```

The following example creates an index on a partitioned table by using page compression on partition `1` of the index and row compression on partitions `2` through `4` of the index.

```
CREATE CLUSTERED INDEX IX_PartTab2Col1
ON PartitionTable1 (Col1)
WITH (DATA_COMPRESSION = PAGE ON PARTITIONS(1),
    DATA_COMPRESSION = ROW ON PARTITIONS (2 TO 4 ) ) ;
GO
```

See Also

ALTER INDEX (Transact-SQL)
CREATE PARTITION FUNCTION
CREATE PARTITION SCHEME
CREATE SPATIAL INDEX (Transact-SQL)
CREATE STATISTICS
CREATE TABLE
CREATE XML INDEX (Transact-SQL)
Data Types
DBCC SHOW_STATISTICS
DROP INDEX
Indexes on xml Type columns
sys.indexes

sys.index_columns
sys.xml_indexes
EVENTDATA (Transact-SQL)

CREATE LOGIN

Creates a Database Engine login for SQL Server, Windows Azure SQL Database, and SQL Server PDW.

Note
The **CREATE LOGIN** options vary for SQL Server, SQL Database, and SQL Server PDW.

Transact-SQL Syntax Conventions

Syntax

```
-- Syntax for SQL Server
CREATE LOGIN login_name { WITH <option_list1> | FROM <sources> }

<option_list1> ::=
    PASSWORD = { 'password' | hashed_password HASHED } [ MUST_CHANGE ]
    [ , <option_list2> [ ,... ] ]

<option_list2> ::=
    SID = sid
    | DEFAULT_DATABASE = database
    | DEFAULT_LANGUAGE = language
    | CHECK_EXPIRATION = { ON | OFF}
    | CHECK_POLICY = { ON | OFF}
    | CREDENTIAL = credential_name

<sources> ::=
    WINDOWS [ WITH <windows_options>  [ ,... ] ]
    | CERTIFICATE certname
    | ASYMMETRIC KEY asym_key_name

<windows_options> ::=
    DEFAULT_DATABASE = database
    | DEFAULT_LANGUAGE = language
```

Syntax

-- Syntax for SQL Database

CREATE LOGIN **login_name** { WITH <option_list3> }

<option_list3> ::=

PASSWORD = { '**password**' }

Syntax

-- Syntax for SQL Server PDW

CREATE LOGIN **login_name** { WITH <option_list4> }

< option_list4> ::=

PASSWORD = { '**password**' } [MUST_CHANGE]

[**,** <option_list5> [**,**...]]

<option_list5> ::=

CHECK_EXPIRATION = { ON | OFF}

| CHECK_POLICY = { ON | OFF}

Arguments

login_name

Specifies the name of the login that is created. There are four types of logins: SQL Server logins, Windows logins, certificate-mapped logins, and asymmetric key-mapped logins. When you are creating logins that are mapped from a Windows domain account, you must use the pre-Windows 2000 user logon name in the format [<domainName>\<login_name>]. You cannot use a UPN in the format login_name@DomainName. For an example, see example D later in this topic. SQL Server authentication logins are type **sysname** and must conform to the rules for Identifiers and cannot contain a '****'. Windows logins can contain a '****'.

PASSWORD = 'password'

Applies to SQL Server logins only. Specifies the password for the login that is being created. You should use a strong password. For more information see Strong Passwords and Password Policy.

Passwords are case-sensitive. Passwords should always be at least 8 characters long, and cannot exceed 128 characters. Passwords can include a-z, A-Z, 0-9, and most non-

alphanumeric characters. Passwords cannot contain single quotes, or the login_name.

PASSWORD = hashed_password

Applies to the HASHED keyword only. Specifies the hashed value of the password for the login that is being created.

HASHED

Applies to SQL Server logins only. Specifies that the password entered after the PASSWORD argument is already hashed. If this option is not selected, the string entered as password is hashed before it is stored in the database. This option should only be used for migrating databases from one server to another. Do not use the HASHED option to create new logins.

MUST_CHANGE

Applies to SQL Server logins only. If this option is included, SQL Server prompts the user for a new password the first time the new login is used.

CREDENTIAL = credential_name

The name of a credential to be mapped to the new SQL Server login. The credential must already exist in the server. Currently this option only links the credential to a login. A credential cannot be mapped to the sa login.

SID = sid

Applies to SQL Server logins only. Specifies the GUID of the new SQL Server login. If this option is not selected, SQL Server automatically assigns a GUID.

DEFAULT_DATABASE = database

Specifies the default database to be assigned to the login. If this option is not included, the default database is set to master.

DEFAULT_LANGUAGE = language

Specifies the default language to be assigned to the login. If this option is not included, the default language is set to the current default language of the server. If the default language of the server is later changed, the default language of the login remains unchanged.

CHECK_EXPIRATION = { ON | OFF }

Applies to SQL Server logins only. Specifies whether password expiration policy should be enforced on this login. The default value is OFF.

CHECK_POLICY = { ON | OFF }

Applies to SQL Server logins only. Specifies that the Windows password policies of the computer on which SQL Server is running should be enforced on this login. The default value is ON.

If the Windows policy requires strong passwords, passwords must contain at least three of the following four characteristics:

- An uppercase character (A-Z).

- A lowercase character (a-z).
- A digit (0-9).
- One of the non-alphanumeric characters, such as a space, _, @, *, ^, %, !, $, #, or &.

WINDOWS

Specifies that the login be mapped to a Windows login.

CERTIFICATE certname

Specifies the name of a certificate to be associated with this login. This certificate must already occur in the master database.

ASYMMETRIC KEY asym_key_name

Specifies the name of an asymmetric key to be associated with this login. This key must already occur in the master database.

Remarks

Passwords are case-sensitive.

Prehashing of passwords is supported only when you are creating SQL Server logins.

If MUST_CHANGE is specified, CHECK_EXPIRATION and CHECK_POLICY must be set to ON. Otherwise, the statement will fail.

A combination of CHECK_POLICY = OFF and CHECK_EXPIRATION = ON is not supported.

When CHECK_POLICY is set to OFF, lockout_time is reset and CHECK_EXPIRATION is set to OFF.

Important
CHECK_EXPIRATION and CHECK_POLICY are only enforced on Windows Server 2003 and later. For more information, see Password Policy.

Logins created from certificates or asymmetric keys are used only for code signing. They cannot be used to connect to SQL Server. You can create a login from a certificate or asymmetric key only when the certificate or asymmetric key already exists in master.

For a script to transfer logins, see How to transfer the logins and the passwords between instances of SQL Server 2005 and SQL Server 2008.

Creating a login automatically enables the new login and grants the login the server level **CONNECT SQL** permission.

SQL Database Logins

In SQL Database, the **CREATE LOGIN** statement must be the only statement in a batch.

In some methods of connecting to SQL Database, such as **sqlcmd**, you must append the SQL Database server name to the login name in the connection string by using the <login>@<server> notation. For example, if your login is `login1` and the fully qualified name of the SQL Database server is `servername.database.windows.net`, the username parameter of the connection string should be `login1@servername`. Because the total length of the username parameter is 128 characters, login_name is limited to 127 characters minus the length of the

server name. In the example, `login_name` can only be 117 characters long because `servername` is 10 characters.

In SQL Database you must be connected to the master database to create a login.

For more information about SQL Database logins, see Managing Databases and Logins in Windows Azure SQL Database.

Permissions

In SQL Server and SQL Server PDW, requires **ALTER ANY LOGIN** permission on the server or membership in the **securityadmin** fixed server role.

In SQL Database, only the server-level principal login (created by the provisioning process) or members of the `loginmanager` database role in the master database can create new logins.

If the **CREDENTIAL** option is used, also requires **ALTER ANY CREDENTIAL** permission on the server.

Next Steps

After creating a login, the login can connect to the Database Engine, SQL Database, or SQL Server PDW appliance, but only has the permissions granted to the **public** role. Consider performing the some of the following activities.

- To connect to a database, create a database user for the login. For more information, see CREATE USER (Transact-SQL).
- Create a user-defined server role by using CREATE SERVER ROLE (Transact-SQL). Use **ALTER SERVER ROLE** ... **ADD MEMBER** to add the new login to the user-defined server role. For more information, see CREATE SERVER ROLE (Transact-SQL) and ALTER SERVER ROLE (Transact-SQL).
- Use **sp_addsrvrolemember** to add the login to a fixed server role. For more information, see Server-Level Roles and sp_addsrvrolemember (Transact-SQL).
- Use the **GRANT** statement, to grant server-level permissions to the new login or to a role containing the login. For more information, see GRANT (Transact-SQL).

Examples

A. Creating a login with a password

Applies to all.

The following example creates a login for a particular user and assigns a password.

```
CREATE LOGIN <login_name> WITH PASSWORD = '<enterStrongPasswordHere>';
GO
```

B. Creating a login with a password

Applies to SQL Server and Advanced Data Warehouse.

The following example creates a login for a particular user and assigns a password. The MUST_CHANGE option requires users to change this password the first time they connect to the server.

```
CREATE LOGIN <login_name> WITH PASSWORD = '<enterStrongPasswordHere>'
MUST_CHANGE;
GO
```

C. Creating a login mapped to a credential

Applies to SQL Server.

The following example creates the login for a particular user, using the user. This login is mapped to the credential.

```
CREATE LOGIN <login_name> WITH PASSWORD = '<enterStrongPasswordHere>',
    CREDENTIAL = <credentialName>;
GO
```

D. Creating a login from a certificate

Applies to SQL Server.

The following example creates login for a particular user from a certificate in master.

```
USE MASTER;
CREATE CERTIFICATE <certificateName>
    WITH SUBJECT = '<login_name> certificate in master database',
    EXPIRY_DATE = '12/05/2025';
GO
CREATE LOGIN <login_name> FROM CERTIFICATE <certificateName>;
GO
```

E. Creating a login from a Windows domain account

Applies to SQL Server.

The following example creates a login from a Windows domain account.

```
CREATE LOGIN [<domainName>\<login_name>] FROM WINDOWS;
GO
```

See Also

Principals

Password Policy

ALTER LOGIN (Transact-SQL)

DROP LOGIN (Transact-SQL)

EVENTDATA (Transact-SQL)

How to: Create A SQL Server Login

CREATE MASTER KEY

Creates a database master key.

 Transact-SQL Syntax Conventions

Syntax

```
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'password'
```

Arguments

PASSWORD = 'password'

Is the password that is used to encrypt the master key in the database. password must meet the Windows password policy requirements of the computer that is running the instance of SQL Server.

Remarks

The database master key is a symmetric key used to protect the private keys of certificates and asymmetric keys that are present in the database. When it is created, the master key is encrypted by using the AES_256 algorithm and a user-supplied password. To enable the automatic decryption of the master key, a copy of the key is encrypted by using the service master key and stored in both the database and in master. Typically, the copy stored in master is silently updated whenever the master key is changed. This default can be changed by using the DROP ENCRYPTION BY SERVICE MASTER KEY option of ALTER MASTER KEY. A master key that is not encrypted by the service master key must be opened by using the OPEN MASTER KEY statement and a password.

The is_master_key_encrypted_by_server column of the sys.databases catalog view in master indicates whether the database master key is encrypted by the service master key.

Information about the database master key is visible in the sys.symmetric_keys catalog view.

Important

You should back up the master key by using BACKUP MASTER KEY and store the backup in a secure, off-site location.

The service master key and database master keys are protected by using the AES-256 algorithm.

Permissions

Requires CONTROL permission on the database.

Examples

The following example creates a database master key for the `AdventureWorks2012` database. The key is encrypted using the password `23987hxJ#KL95234nl0zBe`.

```
USE AdventureWorks2012;
CREATE MASTER KEY ENCRYPTION BY PASSWORD = '23987hxJ#KL95234nl0zBe';
GO
```

See Also

Encryption Hierarchy
sys.databases (Transact-SQL)
OPEN MASTER KEY (Transact-SQL)
ALTER MASTER KEY (Transact-SQL)
DROP MASTER KEY (Transact-SQL)
CLOSE MASTER KEY (Transact-SQL)
Encryption Hierarchy

CREATE MESSAGE TYPE

Creates a new message type. A message type defines the name of a message and the validation that Service Broker performs on messages that have that name. Both sides of a conversation must define the same message types.

 Transact-SQL Syntax Conventions

Syntax

```
CREATE MESSAGE TYPE message_type_name
   [ AUTHORIZATION owner_name ]
   [ VALIDATION = {  NONE
                   | EMPTY
                   | WELL_FORMED_XML
                   | VALID_XML WITH SCHEMA COLLECTION
                                  schema_collection_name
                  } ]
[ ; ]
```

Arguments

message_type_name

Is the name of the message type to create. A new message type is created in the current database and owned by the principal specified in the AUTHORIZATION clause. Server, database, and schema names cannot be specified. The message_type_name can be up to 128 characters.

AUTHORIZATION owner_name

Sets the owner of the message type to the specified database user or role. When the current user is **dbo** or **sa**, owner_name can be the name of any valid user or role. Otherwise, owner_name must be the name of the current user, the name of a user who the current user has IMPERSONATE permission for, or the name of a role to which the current user belongs. When this clause is omitted, the message type belongs to the current user.

VALIDATION

Specifies how Service Broker validates the message body for messages of this type. When this clause is not specified, validation defaults to NONE.

NONE

Specifies that no validation is performed. The message body can contain data, or it can be NULL.

EMPTY

Specifies that the message body must be NULL.

WELL_FORMED_XML

Specifies that the message body must contain well-formed XML.

VALID_XML WITH SCHEMA COLLECTION schema_collection_name

Specifies that the message body must contain XML that complies with a schema in the specified schema collection The schema_collection_name must be the name of an existing XML schema collection.

Remarks

Service Broker validates incoming messages. When a message contains a message body that does not comply with the validation type specified, Service Broker discards the invalid message and returns an error message to the service that sent the message.

Both sides of a conversation must define the same name for a message type. To help troubleshooting, both sides of a conversation typically specify the same validation for the message type, although Service Broker does not require that both sides of the conversation use the same validation.

A message type can not be a temporary object. Message type names starting with **#** are allowed, but are permanent objects.

Permissions

Permission for creating a message type defaults to members of the **db_ddladmin** or **db_owner** fixed database roles and the **sysadmin** fixed server role.

REFERENCES permission for a message type defaults to the owner of the message type, members of the **db_owner** fixed database role, and members of the **sysadmin** fixed server role.

When the CREATE MESSAGE TYPE statement specifies a schema collection, the user executing the statement must have REFERENCES permission on the schema collection specified.

Examples

A. Creating a message type containing well-formed XML

The following example creates a new message type that contains well-formed XML.

```
CREATE MESSAGE TYPE
  [//Adventure-Works.com/Expenses/SubmitExpense]
  VALIDATION = WELL_FORMED_XML ;
```

B. Creating a message type containing typed XML

The following example creates a message type for an expense report encoded in XML. The example creates an XML schema collection that holds the schema for a simple expense report. The example then creates a new message type that validates messages against the schema.

```
CREATE XML SCHEMA COLLECTION ExpenseReportSchema AS
N'<?xml version="1.0" encoding="UTF-16" ?>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     targetNamespace="http://Adventure-Works.com/schemas/expenseReport"
     xmlns:expense="http://Adventure-Works.com/schemas/expenseReport"
     elementFormDefault="qualified"
   >
    <xsd:complexType name="expenseReportType">
       <xsd:sequence>
         <xsd:element name="EmployeeName" type="xsd:string"/>
         <xsd:element name="EmployeeID" type="xsd:string"/>
         <xsd:element name="ItemDetail"
           type="expense:ItemDetailType" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="ItemDetailType">
      <xsd:sequence>
        <xsd:element name="Date" type="xsd:date"/>
        <xsd:element name="CostCenter" type="xsd:string"/>
        <xsd:element name="Total" type="xsd:decimal"/>
        <xsd:element name="Currency" type="xsd:string"/>
        <xsd:element name="Description" type="xsd:string"/>
```

```
      </xsd:sequence>
    </xsd:complexType>

    <xsd:element name="ExpenseReport" type="expense:expenseReportType"/>

  </xsd:schema>' ;


  CREATE MESSAGE TYPE
    [//Adventure-Works.com/Expenses/SubmitExpense]
    VALIDATION = VALID_XML WITH SCHEMA COLLECTION ExpenseReportSchema ;
```

C. Creating a message type for an empty message

The following example creates a new message type with empty encoding.

```
 CREATE MESSAGE TYPE
     [//Adventure-Works.com/Expenses/SubmitExpense]
     VALIDATION = EMPTY ;
```

D. Creating a message type containing binary data

The following example creates a new message type to hold binary data. Because the message will contain data that is not XML, the message type specifies a validation type of `NONE`. Notice that, in this case, the application that receives a message of this type must verify that the message contains data, and that the data is of the type expected.

```
CREATE MESSAGE TYPE
    [//Adventure-Works.com/Expenses/ReceiptImage]
    VALIDATION = NONE ;
```

See Also

EVENTDATA (Transact-SQL)

DROP MESSAGE TYPE

EVENTDATA

CREATE PARTITION FUNCTION

Creates a function in the current database that maps the rows of a table or index into partitions based on the values of a specified column. Using CREATE PARTITION FUNCTION is the first step in creating a partitioned table or index. In SQL Server 2012, a table or index can have a maximum of 15,000 partitions.

Transact-SQL Syntax Conventions

Syntax

```
CREATE PARTITION FUNCTION partition_function_name ( input_parameter_type )
AS RANGE [ LEFT | RIGHT ]
FOR VALUES ( [ boundary_value [ ,...n ] ] )
[ ; ]
```

Arguments

partition_function_name

Is the name of the partition function. Partition function names must be unique within the database and comply with the rules for identifiers.

input_parameter_type

Is the data type of the column used for partitioning. All data types are valid for use as partitioning columns, except **text**, **ntext**, **image**, **xml**, **timestamp**, **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, alias data types, or CLR user-defined data types.

The actual column, known as a partitioning column, is specified in the CREATE TABLE or CREATE INDEX statement.

boundary_value

Specifies the boundary values for each partition of a partitioned table or index that uses partition_function_name. If boundary_value is empty, the partition function maps the whole table or index using partition_function_name into a single partition. Only one partitioning column, specified in a CREATE TABLE or CREATE INDEX statement, can be used.

boundary_value is a constant expression that can reference variables. This includes user-defined type variables, or functions and user-defined functions. It cannot reference Transact-SQL expressions. boundary_value must either match or be implicitly convertible to the data type supplied in input_parameter_type, and cannot be truncated during implicit conversion in a way that the size and scale of the value does not match that of its corresponding input_parameter_type.

Note

If boundary_value consists of **datetime** or **smalldatetime** literals, these literals are evaluated assuming that us_english is the session language. This behavior is deprecated. To make sure the partition function definition behaves as expected for all session languages, we recommend that you use constants that are interpreted the same way for all language settings, such as the yyyymmdd format; or explicitly convert literals to a specific style. To determine the language session of your server, run `SELECT @@LANGUAGE`.

...n

Specifies the number of values supplied by boundary_value, not to exceed 14,999. The number of partitions created is equal to n + 1. The values do not have to be listed in order. If the values are not in order, the Database Engine sorts them, creates the function, and returns a warning that the values are not provided in order. The Database Engine returns an error if n includes any duplicate values.

LEFT | RIGHT

Specifies to which side of each boundary value interval, left or right, the boundary_value [**,**...n] belongs, when interval values are sorted by the Database Engine in ascending order from left to right. If not specified, LEFT is the default.

Remarks

The scope of a partition function is limited to the database that it is created in. Within the database, partition functions reside in a separate namespace from the other functions.

Any rows whose partitioning column has null values are placed in the left-most partition, unless NULL is specified as a boundary value and RIGHT is indicated. In this case, the left-most partition is an empty partition, and NULL values are placed in the following partition.

Permissions

Any one of the following permissions can be used to execute CREATE PARTITION FUNCTION:

- ALTER ANY DATASPACE permission. This permission defaults to members of the **sysadmin** fixed server role and the **db_owner** and **db_ddladmin** fixed database roles.
- CONTROL or ALTER permission on the database in which the partition function is being created.
- CONTROL SERVER or ALTER ANY DATABASE permission on the server of the database in which the partition function is being created.

Examples

A. Creating a RANGE LEFT partition function on an int column

The following partition function will partition a table or index into four partitions.

```
CREATE PARTITION FUNCTION myRangePF1 (int)
AS RANGE LEFT FOR VALUES (1, 100, 1000);
```

The following table shows how a table that uses this partition function on partitioning column **col1** would be partitioned.

Partition	**1**	**2**	**3**	**4**
Values	**col1** <= 1	**col1** > 1 AND **col1** <= 100	**col1** > 100 AND **col1** <= 1000	**col1** > 1000

B. Creating a RANGE RIGHT partition function on an int column

The following partition function uses the same values for boundary_value [**,**...n] as the previous example, except it specifies RANGE RIGHT.

```
CREATE PARTITION FUNCTION myRangePF2 (int)
AS RANGE RIGHT FOR VALUES (1, 100, 1000);
```

The following table shows how a table that uses this partition function on partitioning column **col1** would be partitioned.

Partition	1	2	3	4
Values	**col1** < 1	**col1** >= 1 AND **col1** < 100	**col1** >= 100 AND **col1** < 1000	**col1** >= 1000

C. Creating a RANGE RIGHT partition function on a datetime column

The following partition function partitions a table or index into 12 partitions, one for each month of a year's worth of values in a **datetime** column.

```
CREATE PARTITION FUNCTION [myDateRangePF1] (datetime)
AS RANGE RIGHT FOR VALUES ('20030201', '20030301', '20030401',
               '20030501', '20030601', '20030701', '20030801',
               '20030901', '20031001', '20031101', '20031201');
```

The following table shows how a table or index that uses this partition function on partitioning column **datecol** would be partitioned.

Partition	1	2	...	11	12
Values	**datecol** < February 1, 2003	**datecol** >= February 1, 2003 AND **datecol** < March 1, 2003		**datecol** >= November 1, 2003 AND **col1** < December 1, 2003	**col1** >= December 1, 2003

D. Creating a partition function on a char column

The following partition function partitions a table or index into four partitions.

```
CREATE PARTITION FUNCTION myRangePF3 (char(20))
AS RANGE RIGHT FOR VALUES ('EX', 'RXE', 'XR');
```

The following table shows how a table that uses this partition function on partitioning column **col1** would be partitioned.

Partition	1	2	3	4
Values	**col1** < EX...	**col1** >= EX AND **col1** < RXE...	**col1** >= RXE AND **col1** < XR...	**col1** >= XR

E. Creating 15,000 partitions

The following partition function partitions a table or index into 15,000 partitions.

```
--Create integer partition function for 15,000 partitions.
DECLARE @IntegerPartitionFunction nvarchar(max) = N'CREATE PARTITION FUNCTION
IntegerPartitionFunction (int) AS RANGE RIGHT FOR VALUES (';
DECLARE @i int = 1;
WHILE @i < 14999
BEGIN
    SET @IntegerPartitionFunction += CAST(@i as nvarchar(10)) + N', ';
    SET @i += 1;
END
SET @IntegerPartitionFunction += CAST(@i as nvarchar(10)) + N');';
EXEC sp_executesql @IntegerPartitionFunction;
GO
```

F. Creating partitions for multiple years

The following partition function partitions a table or index into 50 partitions on a **datetime2** column. There is one partitions for each month between January 2007 and January 2011.

```
--Create date partition function with increment by month.
DECLARE @DatePartitionFunction nvarchar(max) = N'CREATE PARTITION FUNCTION
DatePartitionFunction (datetime2) AS RANGE RIGHT FOR VALUES (';
DECLARE @i datetime2 = '20070101';
WHILE @i < '20110101'
BEGIN
    SET @DatePartitionFunction += '''' + CAST(@i as nvarchar(10)) + '''' +
N', ';
    SET @i = DATEADD(MM, 1, @i);
END
SET @DatePartitionFunction += '''' + CAST(@i as nvarchar(10))+ '''' + N');';
```

```
EXEC sp_executesql @DatePartitionFunction;
GO
```

See Also

Partitioned Tables and Indexes
$partition
ALTER PARTITION FUNCTION
DROP PARTITION FUNCTION
CREATE PARTITION SCHEME
CREATE TABLE
CREATE INDEX
ALTER INDEX
EVENTDATA
sys.partition_functions
sys.partition_parameters
sys.partition_range_values
sys.partitions
sys.tables
sys.indexes
sys.index_columns

CREATE PARTITION SCHEME

Creates a scheme in the current database that maps the partitions of a partitioned table or index to filegroups. The number and domain of the partitions of a partitioned table or index are determined in a partition function. A partition function must first be created in a CREATE PARTITION FUNCTION statement before creating a partition scheme.

 Transact-SQL Syntax Conventions

Syntax

CREATE PARTITION SCHEME **partition_scheme_name**
AS PARTITION **partition_function_name**
[ALL] TO **(** { **file_group_name** | **[** PRIMARY **]** } [**,...n**] **)**
[;]

Arguments

partition_scheme_name

Is the name of the partition scheme. Partition scheme names must be unique within the

database and comply with the rules for identifiers.

partition_function_name

Is the name of the partition function using the partition scheme. Partitions created by the partition function are mapped to the filegroups specified in the partition scheme. partition_function_name must already exist in the database. A single partition cannot contain both FILESTREAM and non-FILESTREAM filegroups.

ALL

Specifies that all partitions map to the filegroup provided in file_group_name, or to the primary filegroup if **[**PRIMARY**]** is specified. If ALL is specified, only one file_group_name can be specified.

file_group_name | [PRIMARY] [,...n]

Specifies the names of the filegroups to hold the partitions specified by partition_function_name. file_group_name must already exist in the database.

If **[**PRIMARY**]** is specified, the partition is stored on the primary filegroup. If ALL is specified, only one file_group_name can be specified. Partitions are assigned to filegroups, starting with partition 1, in the order in which the filegroups are listed in [**,**...n]. The same file_group_name can be specified more than one time in [**,**...n]. If n is not sufficient to hold the number of partitions specified in partition_function_name, CREATE PARTITION SCHEME fails with an error.

If partition_function_name generates less partitions than filegroups, the first unassigned filegroup is marked NEXT USED, and an information message displays naming the NEXT USED filegroup. If ALL is specified, the sole file_group_name maintains its NEXT USED property for this partition_function_name. The NEXT USED filegroup will receive an additional partition if one is created in an ALTER PARTITION FUNCTION statement. To create additional unassigned filegroups to hold new partitions, use ALTER PARTITION SCHEME.

When you specify the primary filegroup in file_group_name [1**,**...n], PRIMARY must be delimited, as in **[**PRIMARY**]**, because it is a keyword.

Permissions

The following permissions can be used to execute CREATE PARTITION SCHEME:

- ALTER ANY DATASPACE permission. This permission defaults to members of the **sysadmin** fixed server role and the **db_owner** and **db_ddladmin** fixed database roles.
- CONTROL or ALTER permission on the database in which the partition scheme is being created.
- CONTROL SERVER or ALTER ANY DATABASE permission on the server of the database in which the partition scheme is being created.

Examples

A. Creating a partition scheme that maps each partition to a different filegroup

The following example creates a partition function to partition a table or index into four partitions. A partition scheme is then created that specifies the filegroups to hold each one of the four partitions. This example assumes the filegroups already exist in the database.

```
CREATE PARTITION FUNCTION myRangePF1 (int)
AS RANGE LEFT FOR VALUES (1, 100, 1000);
GO
CREATE PARTITION SCHEME myRangePS1
AS PARTITION myRangePF1
TO (test1fg, test2fg, test3fg, test4fg);
```

The partitions of a table that uses partition function `myRangePF1` on partitioning column **col1** would be assigned as shown in the following table.

Filegroup	`test1fg`	`test2fg`	`test3fg`	`test4fg`
Partition	1	2	3	4
Values	**col1** <= `1`	**col1** > `1` AND **col1** <= `100`	**col1** > `100` AND **col1** <= `1000`	**col1** > `1000`

B. Creating a partition scheme that maps multiple partitions to the same filegroup

If all the partitions map to the same filegroup, use the ALL keyword. But if multiple, but not all, partitions are mapped to the same filegroup, the filegroup name must be repeated, as shown in the following example.

```
CREATE PARTITION FUNCTION myRangePF2 (int)
AS RANGE LEFT FOR VALUES (1, 100, 1000);
GO
CREATE PARTITION SCHEME myRangePS2
AS PARTITION myRangePF2
TO ( test1fg, test1fg, test1fg, test2fg );
```

The partitions of a table that uses partition function `myRangePF2` on partitioning column **col1** would be assigned as shown in the following table.

Filegroup	`test1fg`	`test1fg`	`test1fg`	`test2fg`
Partition	1	2	3	4
Values	**col1** <= `1`	**col1** > 1 AND **col1** <= `100`	**col1** > `100` AND **col1** <= `1000`	**col1** > `1000`

C. Creating a partition scheme that maps all partitions to the same filegroup

The following example creates the same partition function as in the previous examples, and a partition scheme is created that maps all partitions to the same filegroup.

```
CREATE PARTITION FUNCTION myRangePF3 (int)
AS RANGE LEFT FOR VALUES (1, 100, 1000);
GO
CREATE PARTITION SCHEME myRangePS3
AS PARTITION myRangePF3
ALL TO ( test1fg );
```

D. Creating a partition scheme that specifies a 'NEXT USED' filegroup

The following example creates the same partition function as in the previous examples, and a partition scheme is created that lists more filegroups than there are partitions created by the associated partition function.

```
CREATE PARTITION FUNCTION myRangePF4 (int)
AS RANGE LEFT FOR VALUES (1, 100, 1000);
GO
CREATE PARTITION SCHEME myRangePS4
AS PARTITION myRangePF4
TO (test1fg, test2fg, test3fg, test4fg, test5fg)
```

Executing the statement returns the following message.

```
Partition scheme 'myRangePS4' has been created successfully. 'test5fg' is
marked as the next used filegroup in partition scheme 'myRangePS4'.
```

If partition function `myRangePF4` is changed to add a partition, filegroup `test5fg` receives the newly created partition.

See Also

sys.index_columns (Transact-SQL)
ALTER PARTITION SCHEME
DROP PARTITION SCHEME
EVENTDATA
Creating Partitioned Tables and Indexes
sys.partition_schemes
sys.data_spaces
sys.destination_data_spaces
sys.partitions

sys.tables

sys.indexes

sys.index_columns

CREATE PROCEDURE

Creates a Transact-SQL or common language runtime (CLR) stored procedure in SQL Server 2012. 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.

 Transact-SQL Syntax Conventions

Syntax

--Transact-SQL Stored Procedure Syntax

```
CREATE { PROC | PROCEDURE } [schema_name.] procedure_name [ ;number ]
    [ { @parameter [ type_schema_name. ] data_type }
        [ VARYING ] [ = default ] [ OUT | OUTPUT ] [READONLY]
    ] [ ,...n ]
[ WITH <procedure_option> [ ,...n ] ]
[ FOR REPLICATION ]
AS { [ BEGIN ] sql_statement [;] [ ...n ] [ END ] }
[;]

<procedure_option> ::=
    [ ENCRYPTION ]
    [ RECOMPILE ]
    [ EXECUTE AS Clause ]
```

--CLR Stored Procedure Syntax

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

Arguments

schema_name

The name of the schema to which the procedure belongs. Procedures are schema-bound. If a schema name is not 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.

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 cannot be specified for CLR procedures.

The complete name for a procedure or a global temporary procedure, including ##, cannot exceed 128 characters. The complete name for a local temporary procedure, including #, cannot exceed 116 characters.

; number

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 Microsoft SQL Server. Avoid using this feature in new development work, and plan to modify applications that currently use this feature.

Numbered procedures cannot use the **xml** or CLR user-defined types and cannot be used in a plan guide.

@ parameter

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 cannot be used instead of table names, column names, or the names of other database objects. For more information, see EXECUTE.

Parameters cannot 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 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 SQL Data Types and Their .NET Equivalents. For more information about SQL Server system data types and their syntax, see Data Types.
- Table-valued or **cursor** data types cannot 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 is not 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. See Example C below.

Default values are recorded in the **sys.parameters.default** column only for CLR procedures. That column will be 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 cannot 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 cannot be specified as an OUTPUT parameter of a procedure.

READONLY

Indicates that the parameter cannot 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 does not 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 Recompiling Stored Procedures. This option cannot 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 Hint (Transact-SQL).

ENCRYPTION

Indicates that SQL Server will convert the original text of the CREATE PROCEDURE statement to an obfuscated format. The output of the obfuscation is not directly visible in any of the catalog views in SQL Server. Users who have no access to system tables or database files cannot retrieve the obfuscated text. However, the text will be 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 is not valid for CLR procedures.

Procedures created with this option cannot be published as part of SQL Server replication.

EXECUTE AS

Specifies the security context under which to execute the procedure.

For more information, see EXECUTE AS Clause (Transact-SQL).

FOR REPLICATION

Specifies that the procedure is created for replication. Consequently, it cannot 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 cannot be declared if FOR REPLICATION is specified. FOR REPLICATION cannot be specified for CLR procedures. The RECOMPILE option is ignored for procedures created with FOR REPLICATION.

A FOR REPLICATION procedure will have 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

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 cannot execute CLR code. You can create, modify, and drop database objects that reference common language runtime modules; however, you cannot 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.

Best Practices

Although this is not 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. Overall performance of the database and application is improved by eliminating this unnecessary network overhead. For information, see SET NOCOUNT (Transact-SQL).
- Use schema names when creating or referencing database objects in the procedure. It will take less processing time for the Database Engine to resolve object names if it does not have to search multiple schemas. It will also prevent 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 will succeed 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/END 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 will prevent passing NULL to columns that do not 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 are not 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 is not 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 are not filtered out of the result set.

General 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 cannot have any input or output parameters. For more information, see Executing Stored Procedures (Database Engine).

Procedures are nested when one procedure call 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 do not 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 are not 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 cannot be combined with other Transact-SQL statements in a single batch.

The following statements cannot be used anywhere in the body of a stored procedure.

CREATE AGGREGATE	CREATE SCHEMA	SET SHOWPLAN_TEXT
CREATE DEFAULT	CREATE or ALTER TRIGGER	SET SHOWPLAN_XML
CREATE or ALTER FUNCTION	CREATE or ALTER VIEW	USE database_name
CREATE or ALTER PROCEDURE	SET PARSEONLY	
CREATE RULE	SET SHOWPLAN_ALL	

A procedure can reference tables that do not yet exist. At creation time, only syntax checking is performed. The procedure is not 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 do not exist can be created successfully; however, the procedure will fail at execution time if the referenced tables do not exist.

You cannot 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.

```
-- 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 cannot be rolled back. Remote procedures do not 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 SQL Data Types and Their .NET Equivalents.

Metadata

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

View	Description
sys.sql_modules	Returns the definition of a Transact-SQL procedure. The text of a procedure created with the ENCRYPTION option cannot be viewed by using the **sys.sql_modules** 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.

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

*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.

Security

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.

Examples

Category	Featured syntax elements
Basic Syntax	CREATE PROCEDURE
Passing parameters	@parameter • = default • OUTPUT • table-valued parameter type • CURSOR VARYING
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. Creating a simple 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. This procedure does not use any parameters. The example then demonstrates three methods of executing the procedure.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID ( 'HumanResources.uspGetAllEmployees', 'P' ) IS NOT NULL
    DROP PROCEDURE HumanResources.uspGetAllEmployees;
GO
CREATE PROCEDURE HumanResources.uspGetAllEmployees
AS
    SET NOCOUNT ON;
```

```
    SELECT LastName, FirstName, Department
    FROM HumanResources.vEmployeeDepartmentHistory;
GO
```

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

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

B. Returning more than one result set

The following procedure returns two result sets.

```
USE AdventureWorks2012;
GO
CREATE PROCEDURE dbo.uspMultipleResults
AS
SELECT TOP(10) BusinessEntityID, Lastname, FirstName FROM Person.Person;
SELECT TOP(10) CustomerID, AccountNumber FROM Sales.Customer;
GO
```

C. Creating 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.

```
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
```

Passing Parameters

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

A. Creating 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.

```
USE AdventureWorks2012;
GO
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, Department
    FROM HumanResources.vEmployeeDepartmentHistory
    WHERE FirstName = @FirstName AND LastName = @LastName;
GO
```

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

```
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';
```

B. Using 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`).

```
USE AdventureWorks2012;
GO
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, Department
    FROM HumanResources.vEmployeeDepartmentHistory
    WHERE FirstName LIKE @FirstName AND LastName LIKE @LastName;
GO
```

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

```
Copy Code
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%';
```

C. Using OUTPUT parameters

The following example creates the `uspGetList` procedure. This procedures returns a list of products that have prices that do not 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.

```
USE AdventureWorks2012;
GO
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 @ListPprice.
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.

```
USE AdventureWorks2012;
GO
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 @ListPprice.
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
```

Here is the partial result set:

```
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.
```

D. Using 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.

```
USE AdventureWorks2012;
GO

/* 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 [AdventureWorks2012].[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
    [AdventureWorks2012].[Person].[StateProvince];

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

E. Using 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:

```
USE AdventureWorks2012;
GO
IF OBJECT_ID ( 'dbo.uspCurrencyCursor', 'P' ) IS NOT NULL
    DROP PROCEDURE dbo.uspCurrencyCursor;
GO
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.

```
USE AdventureWorks2012;
GO
DECLARE @MyCursor CURSOR;
EXEC dbo.uspCurrencyCursor @CurrencyCursor = @MyCursor OUTPUT;
WHILE (@@FETCH_STATUS = 0)
```

```
BEGIN;
     FETCH NEXT FROM @MyCursor;
END;
CLOSE @MyCursor;
DEALLOCATE @MyCursor;
GO
```

Modifying 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.

A. Using 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`.

```
USE AdventureWorks2012;
GO
CREATE PROCEDURE HumanResources.Update_VacationHours
@NewHours smallint
AS
SET NOCOUNT ON;
UPDATE HumanResources.Employee
SET VacationHours =
    ( CASE
         WHEN SalariedFlag = 0 THEN VacationHours + @NewHours
         ELSE @NewHours
       END
    )
WHERE CurrentFlag = 1;
GO

EXEC HumanResources.Update_VacationHours 40;
```

Error Handling

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

Using TRY...CATCH

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

```
USE AdventureWorks2012;
GO

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;

/* 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;

DROP PROCEDURE Production.uspDeleteWorkOrder;
```

Obfuscating the Procedure Definition

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

A. Using the WITH ENCRYPTION option

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

```
USE AdventureWorks2012;
GO
IF OBJECT_ID ( 'HumanResources.uspEncryptThis', 'P' ) IS NOT NULL
    DROP PROCEDURE HumanResources.uspEncryptThis;
GO
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`:

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

Here is the result set.

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

Directly query the `sys.sql_modules` catalog view:

```
USE AdventureWorks2012;
GO
SELECT definition FROM sys.sql_modules
WHERE object_id = OBJECT_ID('HumanResources.uspEncryptThis');
```

Here is the result set.

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

Forcing the Procedure to Recompile

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

A. Using the WITH RECOMPILE option

The `WITH RECOMPILE` clause is helpful when the parameters supplied to the procedure will not be typical, and when a new execution plan should not be cached or stored in memory.

```
USE AdventureWorks2012;
GO
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;
GO
```

Setting the Security Context

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

A. Using 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.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID ( 'Purchasing.uspVendorAllInfo', 'P' ) IS NOT NULL
    DROP PROCEDURE Purchasing.uspVendorAllInfo;
GO
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
```

B. Creating custom permission sets

The following example uses EXECUTE AS to create custom permissions for a database operation. Some operations such as TRUNCATE TABLE, do not 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.

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

See Also

ALTER PROCEDURE
Control-of-Flow Language
Cursors (Database Engine)
Data Types (Transact-SQL)
DECLARE @local_variable
DROP PROCEDURE
EXECUTE (Transact-SQL)
EXECUTE AS (Transact-SQL)
Stored Procedures (Database Engine)
sp_procoption
sp_recompile
sys.sql_modules
sys.parameters
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)
How to: Create a stored procedure (SQL Server Management Studio)
Table-valued Parameters (Database Engine)
sys.dm_sql_referenced_entities
sys.dm_sql_referencing_entities

CREATE QUEUE

Creates a new queue in a database. Queues store messages. When a message arrives for a service, Service Broker puts the message on the queue associated with the service.

 Transact-SQL Syntax Conventions

Syntax

```
CREATE QUEUE <object>
  [ WITH
    [ STATUS = { ON | OFF }  [ , ] ]
    [ RETENTION = { ON | OFF } [ , ] ]
    [ ACTIVATION (
```

```
      [ STATUS = { ON | OFF } , ]
        PROCEDURE_NAME = <procedure> ,
        MAX_QUEUE_READERS = max_readers ,
        EXECUTE AS { SELF | 'user_name' | OWNER }
        ) [ , ] ]
    [ POISON_MESSAGE_HANDLING (
      [ STATUS = { ON | OFF } )
   ]
    [ ON { filegroup | [ DEFAULT ] } ]
[ ; ]

<object> ::=
{
  [ database_name. [ schema_name ] . | schema_name. ]
    queue_name
}

<procedure> ::=
{
  [ database_name. [ schema_name ] . | schema_name. ]
    stored_procedure_name
}
```

Arguments

database_name (object)

Is the name of the database within which to create the new queue. database_name must specify the name of an existing database. When database_name is not provided, the queue is created in the current database.

schema_name (object)

Is the name of the schema to which the new queue belongs. The schema defaults to the default schema for the user that executes the statement. If the CREATE QUEUE statement is executed by a member of the sysadmin fixed server role, or a member of the db_dbowner or db_ddladmin fixed database roles in the database specified by database_name, schema_name can specify a schema other than the one associated with the login of the current connection. Otherwise, schema_name must be the default schema for the user who executes the statement.

queue_name

Is the name of the queue to create. This name must meet the guidelines for SQL Server identifiers.

STATUS (Queue)

Specifies whether the queue is available (ON) or unavailable (OFF). When the queue is unavailable, no messages can be added to the queue or removed from the queue. You can create the queue in an unavailable state to keep messages from arriving on the queue until the queue is made available with an ALTER QUEUE statement. If this clause is omitted, the default is ON, and the queue is available.

RETENTION

Specifies the retention setting for the queue. If RETENTION = ON, all messages sent or received on conversations that use this queue are retained in the queue until the conversations have ended. This lets you retain messages for auditing purposes, or to perform compensating transactions if an error occurs. If this clause is not specified, the retention setting defaults to OFF.

Note

Setting RETENTION = ON can decrease performance. This setting should only be used if it is required for the application.

ACTIVATION

Specifies information about which stored procedure you have to start to process messages in this queue.

STATUS (Activation)

Specifies whether Service Broker starts the stored procedure. When STATUS = ON, the queue starts the stored procedure specified with PROCEDURE_NAME when the number of procedures currently running is less than MAX_QUEUE_READERS and when messages arrive on the queue faster than the stored procedures receive messages. When STATUS = OFF, the queue does not start the stored procedure. If this clause is not specified, the default is ON.

PROCEDURE_NAME = <procedure>

Specifies the name of the stored procedure to start to process messages in this queue. This value must be a SQL Server identifier.

database_name(procedure)

Is the name of the database that contains the stored procedure.

schema_name(procedure)

Is the name of the schema that contains the stored procedure.

procedure_name

Is the name of the stored procedure.

MAX_QUEUE_READERS = max_readers

Specifies the maximum number of instances of the activation stored procedure that the queue starts at the same time. The value of max_readers must be a number between **0** and **32767**.

EXECUTE AS

Specifies the SQL Server database user account under which the activation stored procedure runs. SQL Server must be able to check the permissions for this user at the time that the queue starts the stored procedure. For a domain user, the server must be connected to the domain when the procedure is started or activation fails. For a SQL Server user, the server can always check permissions.

SELF

Specifies that the stored procedure executes as the current user. (The database principal executing this CREATE QUEUE statement.)

'user_name'

Is the name of the user who the stored procedure executes as. The user_name parameter must be a valid SQL Server user specified as a SQL Server identifier. The current user must have IMPERSONATE permission for the user_name specified.

OWNER

Specifies that the stored procedure executes as the owner of the queue.

POISON_MESSAGE_HANDLING

Specifies whether poison message handling is enabled for the queue. The default is ON.

A queue that has poison message handling set to OFF will not be disabled after five consecutive transaction rollbacks. This allows for a custom poison message handing system to be defined by the application.

ON filegroup | [DEFAULT]

Specifies the SQL Server filegroup on which to create this queue. You can use the filegroup parameter to identify a filegroup, or use the DEFAULT identifier to use the default filegroup for the service broker database. In the context of this clause, DEFAULT is not a keyword, and must be delimited as an identifier. When no filegroup is specified, the queue uses the default filegroup for the database.

Remarks

A queue can be the target of a SELECT statement. However, the contents of a queue can only be modified using statements that operate on Service Broker conversations, such as SEND, RECEIVE, and END CONVERSATION. A queue cannot be the target of an INSERT, UPDATE, DELETE, or TRUNCATE statement.

A queue might not be a temporary object. Therefore, queue names starting with **#** are not valid.

Creating a queue in an inactive state lets you get the infrastructure in place for a service before allowing messages to be received on the queue.

Service Broker does not stop activation stored procedures when there are no messages on the queue. An activation stored procedure should exit when no messages are available on the queue for a short time.

Permissions for the activation stored procedure are checked when Service Broker starts the stored procedure, not when the queue is created. The CREATE QUEUE statement does not verify that the user specified in the EXECUTE AS clause has permission to execute the stored procedure specified in the PROCEDURE NAME clause.

When a queue is unavailable, Service Broker holds messages for services that use the queue in the transmission queue for the database. The sys.transmission_queue catalog view provides a view of the transmission queue.

A queue is a schema-owned object. Queues appear in the sys.objects catalog view.

The following table lists the columns in a queue.

Column name	Data type	Description
status	**tinyint**	Status of the message. The RECEIVE statement returns all messages that have a status of **1**. If message retention is on, the status is then set to 0. If message retention is off, the message is deleted from the queue. Messages in the queue can contain one of the following values: **0**=Retained received message **1**=Ready to receive **2**=Not yet complete **3**=Retained sent message
priority	**tinyint**	The priority level that is assigned to this message.
queuing_order	**bigint**	Message order number in the queue.
conversation_group_id	**uniqueidentifier**	Identifier for the conversation group that

Column name	Data type	Description
		this message belongs to.
conversation_handle	**uniqueidentifier**	Handle for the conversation that this message is part of.
message_sequence_number	**bigint**	Sequence number of the message in the conversation.
service_name	**nvarchar(512)**	Name of the service that the conversation is to.
service_id	**int**	SQL Server object identifier of the service that the conversation is to.
service_contract_name	**nvarchar(256)**	Name of the contract that the conversation follows.
service_contract_id	**int**	SQL Server object identifier of the contract that the conversation follows.
message_type_name	**nvarchar(256)**	Name of the message type that describes the message.
message_type_id	**int**	SQL Server object identifier of the message type that describes the message.
validation	**nchar(2)**	Validation used for the message. E=Empty N=None X=XML
message_body	**varbinary(MAX)**	Content of the message.
message_id	**uniqueidentifier**	Unique identifier for the message.

Permissions

Permission for creating a queue uses members of the db_ddladmin or db_owner fixed database roles and the sysadmin fixed server role.

REFERENCES permission for a queue defaults to the owner of the queue, members of the db_ddladmin or db_owner fixed database roles, and members of the sysadmin fixed server role.

RECEIVE permission for a queue defaults to the owner of the queue, members of the db_owner fixed database role, and members of the sysadmin fixed server role.

Examples

A. Creating a queue with no parameters

The following example creates a queue that is available to receive messages. No activation stored procedure is specified for the queue.

```
CREATE QUEUE ExpenseQueue ;
```

B. Creating an unavailable queue

The following example creates a queue that is unavailable to receive messages. No activation stored procedure is specified for the queue.

```
CREATE QUEUE ExpenseQueue WITH STATUS=OFF ;
```

C. Creating a queue and specify internal activation information

The following example creates a queue that is available to receive messages. The queue starts the stored procedure `expense_procedure` when a message enters the queue. The stored procedure executes as the user `ExpenseUser`. The queue starts a maximum of `5` instances of the stored procedure.

```
CREATE QUEUE ExpenseQueue
    WITH STATUS=ON,
    ACTIVATION (
        PROCEDURE_NAME = expense_procedure,
        MAX_QUEUE_READERS = 5,
        EXECUTE AS 'ExpenseUser' ) ;
```

D. Creating a queue on a specific filegroup

The following example creates a queue on the filegroup `ExpenseWorkFileGroup`.

```
CREATE QUEUE ExpenseQueue
    ON ExpenseWorkFileGroup ;
```

E. Creating a queue with multiple parameters

The following example creates a queue on the `DEFAULT` filegroup. The queue is unavailable. Messages are retained in the queue until the conversation that they belong to ends. When the queue is made available through ALTER QUEUE, the queue starts the stored procedure `2008R2.dbo.expense_procedure` to process messages. The stored procedure executes as the user who ran the `CREATE QUEUE` statement. The queue starts a maximum of `10` instances of the stored procedure.

```
CREATE QUEUE ExpenseQueue
    WITH STATUS = OFF,
      RETENTION = ON,
      ACTIVATION (
          PROCEDURE_NAME = AdventureWorks2012.dbo.expense_procedure,
          MAX_QUEUE_READERS = 10,
          EXECUTE AS SELF )
    ON [DEFAULT] ;
```

See Also

ALTER QUEUE
CREATE SERVICE (Transact-SQL)
DROP QUEUE
RECEIVE (Transact-SQL)
EVENTDATA (Transact-SQL)

CREATE REMOTE SERVICE BINDING

Creates a binding that defines the security credentials to use to initiate a conversation with a remote service.

Transact-SQL Syntax Conventions

Syntax

```
CREATE REMOTE SERVICE BINDING binding_name
  [ AUTHORIZATION owner_name ]
  TO SERVICE 'service_name'
  WITH  USER = user_name [ , ANONYMOUS = { ON | OFF } ]
[ ; ]
```

Arguments

binding_name

Is the name of the remote service binding to be created. Server, database, and schema names cannot be specified. The binding_name must be a valid **sysname**.

AUTHORIZATION owner_name

Sets the owner of the binding to the specified database user or role. When the current user is **dbo** or **sa**, owner_name can be the name of any valid user or role. Otherwise, owner_name must be the name of the current user, the name of a user who the current user has

IMPERSONATE permissions for, or the name of a role to which the current user belongs.

TO SERVICE 'service_name'

Specifies the remote service to bind to the user identified in the WITH USER clause.

USER = user_name

Specifies the database principal that owns the certificate associated with the remote service identified by the TO SERVICE clause. This certificate is used for encryption and authentication of messages exchanged with the remote service.

ANONYMOUS

Specifies whether anonymous authentication is used when communicating with the remote service. If ANONYMOUS = ON, anonymous authentication is used and operations in the remote database occur as a member of the **public** fixed database role. If ANONYMOUS = OFF, operations in the remote database occur as a specific user in that database. If this clause is not specified, the default is OFF.

Remarks

Service Broker uses a remote service binding to locate the certificate to use for a new conversation. The public key in the certificate associated with user_name is used to authenticate messages sent to the remote service and to encrypt a session key that is then used to encrypt the conversation. The certificate for user_name must correspond to the certificate for a user in the database that hosts the remote service.

A remote service binding is only necessary for initiating services that communicate with target services outside of the SQL Server instance. A database that hosts an initiating service must contain remote service bindings for any target services outside of the SQL Server instance. A database that hosts a target service need not contain remote service bindings for the initiating services that communicate with the target service. When the initiator and target services are in the same instance of SQL Server, no remote service binding is necessary. However, if a remote service binding is present where the service_name specified for TO SERVICE matches the name of the local service, Service Broker will use the binding.

When ANONYMOUS = ON, the initiating service connects to the target service as a member of the **public** fixed database role. By default, members of this role do not have permission to connect to a database. To successfully send a message, the target database must grant the **public** role CONNECT permission for the database and SEND permission for the target service.

When a user owns more than one certificate, Service Broker selects the certificate with the latest expiration date from among the certificates that currently valid and marked as AVAILABLE FOR BEGIN_DIALOG.

Permissions

Permissions for creating a remote service binding default to the user named in the USER clause, members of the **db_owner** fixed database role, members of the **db_ddladmin** fixed database role, and members of the **sysadmin** fixed server role.

The user that executes the CREATE REMOTE SERVICE BINDING statement must have impersonate permission for the principal specified in the statement.

A remote service binding may not be a temporary object. Remote service binding names beginning with **#** are allowed, but are permanent objects.

Examples

A. Creating a remote service binding

The following example creates a binding for the service `//Adventure-Works.com/services/AccountsPayable`. Service Broker uses the certificate owned by the `APUser` database principal to authenticate to the remote service and to exchange the session encryption key with the remote service.

```
CREATE REMOTE SERVICE BINDING APBinding
    TO SERVICE '//Adventure-Works.com/services/AccountsPayable'
    WITH USER = APUser ;
```

B. Creating a remote service binding using anonymous authentication

The following example creates a binding for the service `//Adventure-Works.com/services/AccountsPayable`. Service Broker uses the certificate owned by the `APUser` database principal to exchange the session encryption key with the remote service. The broker does not authenticate to the remote service. In the database that hosts the remote service, messages are delivered as the **guest** user.

```
CREATE REMOTE SERVICE BINDING APBinding
    TO SERVICE '//Adventure-Works.com/services/AccountsPayable'
    WITH USER = APUser, ANONYMOUS=ON ;
```

See Also

Certificates for Dialog Security
DROP REMOTE SERVICE BINDING
EVENTDATA

CREATE RESOURCE POOL

Creates a Resource Governor resource pool. Resource Governor is not available in every edition of Microsoft SQL Server. For a list of features that are supported by the editions of SQL Server, see Features Supported by the Editions of SQL Server 2012.

 Transact-SQL Syntax Conventions.

Syntax

CREATE RESOURCE POOL **`pool_name`**

[WITH

([MIN_CPU_PERCENT = **value**]

[[,] MAX_CPU_PERCENT = **value**]

[[,] CAP_CPU_PERCENT = **value**]

[[,] AFFINITY {SCHEDULER = AUTO | (Scheduler_range_spec) | NUMANODE = (NUMA_node_range_spec)}]

[[,] MIN_MEMORY_PERCENT = **value**]

[[,] MAX_MEMORY_PERCENT = **value**])

]

[;]

Scheduler_range_spec::=

{SCHED_ID | SCHED_ID TO SCHED_ID}[,...n]

NUMA_node_range_spec::=

{NUMA_node_ID | NUMA_node_ID TO NUMA_node_ID}[,...n]

Arguments

pool_name

Is the user-defined name for the resource pool. pool_name is alphanumeric, can be up to 128 characters, must be unique within an instance of SQL Server, and must comply with the rules for identifiers.

MIN_CPU_PERCENT = value

Specifies the guaranteed average CPU bandwidth for all requests in the resource pool when there is CPU contention. value is an integer with a default setting of 0. The allowed range for value is from 0 through 100.

MAX_CPU_PERCENT = value

Specifies the maximum average CPU bandwidth that all requests in resource pool will receive when there is CPU contention. value is an integer with a default setting of 100. The allowed range for value is from 1 through 100.

CAP_CPU_PERCENT = value

Specifies a hard cap on the CPU bandwidth that all requests in the resource pool will receive. Limits the maximum CPU bandwidth level to be the same as the specified value. value is an integer with a default setting of 100. The allowed range for value is from 1 through 100.

AFFINITY {SCHEDULER = AUTO | (Scheduler_range_spec) | NUMANODE = (<NUMA_node_range_spec>)}

Attach the resource pool to specific schedulers. The default value is AUTO.

MIN_MEMORY_PERCENT = value

Specifies the minimum amount of memory reserved for this resource pool that can not be shared with other resource pools. value is an integer with a default setting of 0 The allowed range for value is from 0 to 100.

MAX_MEMORY_PERCENT = value

Specifies the total server memory that can be used by requests in this resource pool. value is an integer with a default setting of 100. The allowed range for value is from 1 through 100.

Remarks

The values for MAX_CPU_PERCENT and MAX_MEMORY_PERCENT must be greater than or equal to the values for MIN_CPU_PERCENT and MIN_MEMORY_PERCENT, respectively.

CAP_CPU_PERCENT differs from MAX_CPU_PERCENT in that workloads associated with the pool can use CPU capacity above the value of MAX_CPU_PERCENT if it is available, but not above the value of CAP_CPU_PERCENT.

The total CPU percentage for each affinitized component (scheduler(s) or NUMA node(s)) should not exceed 100%.

Permissions

Requires CONTROL SERVER permission.

Examples

The following example shows how to create a resource pool named `bigPool`. This pool uses the default Resource Governor settings.

```
CREATE RESOURCE POOL bigPool;
GO
ALTER RESOURCE GOVERNOR RECONFIGURE;
GO
```

In the following example, the `CAP_CPU_PERCENT` sets the hard cap to 30% and `AFFINITY SCHEDULER` is set to a range of 0 to 63, 128 to 191.

```
CREATE RESOURCE POOL PoolAdmin
WITH (
     MIN_CPU_PERCENT = 10,
     MAX_CPU_PERCENT = 20,
     CAP_CPU_PERCENT = 30,
     AFFINITY SCHEDULER = (0 TO 63, 128 TO 191),
     MIN_MEMORY_PERCENT = 5,
     MAX_MEMORY_PERCENT = 15);
```

Remarks

The Resource Governor feature enables a database administrator to distribute server resources among resource pools, up to a maximum of 64 pools.

See Also

ALTER RESOURCE POOL (Transact-SQL)
DROP RESOURCE POOL (Transact-SQL)
CREATE WORKLOAD GROUP (Transact-SQL)
ALTER WORKLOAD GROUP (Transact-SQL)
DROP WORKLOAD GROUP (Transact-SQL)
ALTER RESOURCE GOVERNOR (Transact-SQL)

CREATE ROLE

Creates a new database role in the current database.

 Transact-SQL Syntax Conventions

Syntax

```
CREATE ROLE role_name [ AUTHORIZATION owner_name ]
```

Arguments

role_name

Is the name of the role to be created.

AUTHORIZATION owner_name

Is the database user or role that is to own the new role. If no user is specified, the role will be owned by the user that executes CREATE ROLE.

Remarks

Roles are database-level securables. After you create a role, configure the database-level permissions of the role by using GRANT, DENY, and REVOKE. To add members to a database role, use ALTER ROLE (Transact-SQL). For more information, see sys.database_principals (Transact-SQL).

Database roles are visible in the sys.database_role_members and sys.database_principals catalog views.

 **Caution**

Beginning with SQL Server 2005, the behavior of schemas changed. As a result, code that assumes that schemas are equivalent to database users may no longer return correct results. Old catalog views, including , should not be used in a database in which any of the following DDL statements have ever been used: CREATE SCHEMA, ALTER SCHEMA, DROP SCHEMA, CREATE USER, ALTER USER, DROP USER, CREATE ROLE, ALTER ROLE, DROP ROLE, CREATE APPROLE, ALTER APPROLE, DROP APPROLE, ALTER AUTHORIZATION. In such databases you must instead use the new catalog views. The new catalog views take into account the separation of principals and schemas that was introduced in SQL Server 2005. For more information about catalog views, see .

Permissions

Requires **CREATE ROLE** permission on the database or membership in the **db_securityadmin** fixed database role. When you use the **AUTHORIZATION** option, the following permissions are also required:

- To assign ownership of a role to another user, requires IMPERSONATE permission on that user.
- To assign ownership of a role to another role, requires membership in the recipient role or ALTER permission on that role.
- To assign ownership of a role to an application role, requires ALTER permission on the application role.

Examples

A. Creating a database role that is owned by a database user

The following example creates the database role `buyers` that is owned by user `BenMiller`.

```
USE AdventureWorks2012;

CREATE ROLE buyers AUTHORIZATION BenMiller;

GO
```

B. Creating a database role that is owned by a fixed database role

The following example creates the database role `auditors` that is owned the `db_securityadmin` fixed database role.

```
USE AdventureWorks2012;

CREATE ROLE auditors AUTHORIZATION db_securityadmin;

GO
```

See Also

Principals

ALTER ROLE (Transact-SQL)

DROP ROLE (Transact-SQL)

EVENTDATA (Transact-SQL)

sp_addrolemember (Transact-SQL)
sys.database_role_members (Transact-SQL)
sys.database_principals (Transact-SQL)

CREATE ROUTE

Adds a new route to the routing table for the current database. For outgoing messages, Service Broker determines routing by checking the routing table in the local database. For messages on conversations that originate in another instance, including messages to be forwarded, Service Broker checks the routes in **msdb**.

 Transact-SQL Syntax Conventions

Syntax

```
CREATE ROUTE route_name
[ AUTHORIZATION owner_name ]
WITH
  [ SERVICE_NAME = 'service_name', ]
  [ BROKER_INSTANCE = 'broker_instance_identifier', ]
  [ LIFETIME = route_lifetime, ]
  ADDRESS = 'next_hop_address'
  [ , MIRROR_ADDRESS = 'next_hop_mirror_address' ]
[ ; ]
```

Arguments

route_name

Is the name of the route to create. A new route is created in the current database and owned by the principal specified in the AUTHORIZATION clause. Server, database, and schema names cannot be specified. The route_name must be a valid **sysname**.

AUTHORIZATION owner_name

Sets the owner of the route to the specified database user or role. The owner_name can be the name of any valid user or role when the current user is a member of either the **db_owner** fixed database role or the **sysadmin** fixed server role. Otherwise, owner_name must be the name of the current user, the name of a user that the current user has IMPERSONATE permission for, or the name of a role to which the current user belongs. When this clause is omitted, the route belongs to the current user.

WITH

Introduces the clauses that define the route being created.

SERVICE_NAME = 'service_name'

Specifies the name of the remote service that this route points to. The service_name must exactly match the name the remote service uses. Service Broker uses a byte-by-byte comparison to match the service_name. In other words, the comparison is case sensitive and does not consider the current collation. If the SERVICE_NAME is omitted, this route matches any service name, but has lower priority for matching than a route that specifies a SERVICE_NAME. A route with a service name of **'SQL/ServiceBroker/BrokerConfiguration'** is a route to a Broker Configuration Notice service. A route to this service might not specify a broker instance.

BROKER_INSTANCE = 'broker_instance_identifier'

Specifies the database that hosts the target service. The broker_instance_identifier parameter must be the broker instance identifier for the remote database, which can be obtained by running the following query in the selected database:

```
SELECT service_broker_guid
FROM sys.databases
WHERE database_id = DB_ID()
```

When the BROKER_INSTANCE clause is omitted, this route matches any broker instance. A route that matches any broker instance has higher priority for matching than routes with an explicit broker instance when the conversation does not specify a broker instance. For conversations that specify a broker instance, a route with a broker instance has higher priority than a route that matches any broker instance.

LIFETIME = route_lifetime

Specifies the time, in seconds, that SQL Server retains the route in the routing table. At the end of the lifetime, the route expires, and SQL Server no longer considers the route when choosing a route for a new conversation. If this clause is omitted, the route_lifetime is NULL and the route never expires.

ADDRESS = 'next_hop_address'

Specifies the network address for this route. The next_hop_address specifies a TCP/IP address in the following format:

TCP://{ *dns_name* | *netbios_name* | *ip_address* } **:** *port_number*

The specified *port_number* must match the port number for the Service Broker endpoint of an instance of SQL Server at the specified computer. This can be obtained by running the following query in the selected database:

```
SELECT tcpe.port
FROM sys.tcp_endpoints AS tcpe
INNER JOIN sys.service_broker_endpoints AS ssbe
   ON ssbe.endpoint_id = tcpe.endpoint_id
WHERE ssbe.name = N'MyServiceBrokerEndpoint';
```

When the service is hosted in a mirrored database, you must also specify the MIRROR_ADDRESS for the other instance that hosts a mirrored database. Otherwise, this route does not fail over to the mirror.

When a route specifies **'LOCAL'** for the next_hop_address, the message is delivered to a service within the current instance of SQL Server.

When a route specifies **'TRANSPORT'** for the next_hop_address, the network address is determined based on the network address in the name of the service. A route that specifies **'TRANSPORT'** might not specify a service name or broker instance.

MIRROR_ADDRESS = 'next_hop_mirror_address'

Specifies the network address for a mirrored database with one mirrored database hosted at the next_hop_address. The next_hop_mirror_address specifies a TCP/IP address in the following format:

TCP://{ *dns_name* | *netbios_name* | *ip_address* } **:** *port_number*

The specified *port_number* must match the port number for the Service Broker endpoint of an instance of SQL Server at the specified computer. This can be obtained by running the following query in the selected database:

```
SELECT tcpe.port
FROM sys.tcp_endpoints AS tcpe
INNER JOIN sys.service_broker_endpoints AS ssbe
   ON ssbe.endpoint_id = tcpe.endpoint_id
WHERE ssbe.name = N'MyServiceBrokerEndpoint';
```

When the MIRROR_ADDRESS is specified, the route must specify the SERVICE_NAME clause and the BROKER_INSTANCE clause. A route that specifies **'LOCAL'** or **'TRANSPORT'** for the next_hop_address might not specify a mirror address.

Remarks

The routing table that stores the routes is a metadata table that can be read through the **sys.routes** catalog view. This catalog view can only be updated through the CREATE ROUTE, ALTER ROUTE, and DROP ROUTE statements.

By default, the routing table in each user database contains one route. This route is named **AutoCreatedLocal**. The route specifies **'LOCAL'** for the next_hop_address and matches any service name and broker instance identifier.

When a route specifies **'TRANSPORT'** for the next_hop_address, the network address is determined based on the name of the service. SQL Server can successfully process service names that begin with a network address in a format that is valid for a next_hop_address.

The routing table can contain any number of routes that specify the same service, network address, and broker instance identifier. In this case, Service Broker chooses a route using a procedure designed to find the most exact match between the information specified in the conversation and the information in the routing table.

Service Broker does not remove expired routes from the routing table. An expired route can be made active using the ALTER ROUTE statement.

A route cannot be a temporary object. Route names that start with **#** are allowed, but are permanent objects.

Permissions

Permission for creating a route defaults to members of the **db_ddladmin** or **db_owner** fixed database roles and the **sysadmin** fixed server role.

Examples

A. Creating a TCP/IP route by using a DNS name

The following example creates a route to the service `//Adventure-Works.com/Expenses`. The route specifies that messages to this service travel over TCP to port `1234` on the host identified by the DNS name `www.Adventure-Works.com`. The target server delivers the messages upon arrival to the broker instance identified by the unique identifier `D8D4D268-00A3-4C62-8F91-634B89C1E315`.

```
CREATE ROUTE ExpenseRoute
    WITH
    SERVICE_NAME = '//Adventure-Works.com/Expenses',
    BROKER_INSTANCE = 'D8D4D268-00A3-4C62-8F91-634B89C1E315',
    ADDRESS = 'TCP://www.Adventure-Works.com:1234' ;
```

B. Creating a TCP/IP route by using a NetBIOS name

The following example creates a route to the service `//Adventure-Works.com/Expenses`. The route specifies that messages to this service travel over TCP to port `1234` on the host identified by the NetBIOS name `SERVER02`. Upon arrival, the target SQL Server delivers the message to the database instance identified by the unique identifier `D8D4D268-00A3-4C62-8F91-634B89C1E315`.

```
CREATE ROUTE ExpenseRoute
    WITH
    SERVICE_NAME = '//Adventure-Works.com/Expenses',
    BROKER_INSTANCE = 'D8D4D268-00A3-4C62-8F91-634B89C1E315',
    ADDRESS = 'TCP://SERVER02:1234' ;
```

C. Creating a TCP/IP route by using an IP address

The following example creates a route to the service `//Adventure-Works.com/Expenses`. The route specifies that messages to this service travel over TCP to port `1234` on the host at the IP address `192.168.10.2`. Upon arrival, the target SQL Server delivers the message to the broker instance identified by the unique identifier `D8D4D268-00A3-4C62-8F91-634B89C1E315`.

```
CREATE ROUTE ExpenseRoute
```

```
    WITH
    SERVICE_NAME = '//Adventure-Works.com/Expenses',
    BROKER_INSTANCE = 'D8D4D268-00A3-4C62-8F91-634B89C1E315',
    ADDRESS = 'TCP://192.168.10.2:1234' ;
```

D. Creating a route to a forwarding broker

The following example creates a route to the forwarding broker on the server `dispatch.Adventure-Works.com`. Because both the service name and the broker instance identifier are not specified, SQL Server uses this route for services that have no other route defined.

```
CREATE ROUTE ExpenseRoute
    WITH
    ADDRESS = 'TCP://dispatch.Adventure-Works.com' ;
```

E. Creating a route to a local service

The following example creates a route to the service `//Adventure-Works.com/LogRequests` in the same instance as the route.

```
CREATE ROUTE LogRequests
    WITH
    SERVICE_NAME = '//Adventure-Works.com/LogRequests',
    ADDRESS = 'LOCAL' ;
```

F. Creating a route with a specified lifetime

The following example creates a route to the service `//Adventure-Works.com/Expenses`. The lifetime for the route is `259200` seconds, which equates to 72 hours.

```
CREATE ROUTE ExpenseRoute
    WITH
    SERVICE_NAME = '//Adventure-Works.com/Expenses',
    LIFETIME = 259200,
    ADDRESS = 'TCP://services.Adventure-Works.com:1234' ;
```

G. Creating a route to a mirrored database

The following example creates a route to the service `//Adventure-Works.com/Expenses`. The service is hosted in a database that is mirrored. One of the mirrored databases is located at the address `services.Adventure-Works.com:1234`, and the other database is located at the address `services-mirror.Adventure-Works.com:1234`.

```
CREATE ROUTE ExpenseRoute
    WITH
```

```
    SERVICE_NAME = '//Adventure-Works.com/Expenses',
    BROKER_INSTANCE = '69fcc80c-2239-4700-8437-1001ecddf933',
    ADDRESS = 'TCP://services.Adventure-Works.com:1234',
    MIRROR_ADDRESS = 'TCP://services-mirror.Adventure-Works.com:1234' ;
```

H. Creating a route that uses the service name for routing

The following example creates a route that uses the service name to determine the network address to send the message to. Notice that a route that specifies `'TRANSPORT'` as the network address has lower priority for matching than other routes.

```
CREATE ROUTE TransportRoute
    WITH ADDRESS = 'TRANSPORT' ;
```

See Also

ALTER ROUTE
DROP ROUTE
EVENTDATA

CREATE RULE

Creates an object called a rule. When bound to a column or an alias data type, a rule specifies the acceptable values that can be inserted into that column.

Important
This feature will be removed in a future version of Microsoft SQL Server. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. We recommend that you use check constraints instead. Check constraints are created by using the CHECK keyword of CREATE TABLE or ALTER TABLE. For more information, see Unique Constraints and Check Constraints.

A column or alias data type can have only one rule bound to it. However, a column can have both a rule and one or more check constraints associated with it. When this is true, all restrictions are evaluated.

Transact-SQL Syntax Conventions

Syntax

CREATE RULE [**schema_name .**] **rule_name**
AS **condition_expression**
[;]

Arguments

schema_name

Is the name of the schema to which the rule belongs.

rule_name

Is the name of the new rule. Rule names must comply with the rules for identifiers. Specifying the rule owner name is optional.

condition_expression

Is the condition or conditions that define the rule. A rule can be any expression valid in a WHERE clause and can include elements such as arithmetic operators, relational operators, and predicates (for example, IN, LIKE, BETWEEN). A rule cannot reference columns or other database objects. Built-in functions that do not reference database objects can be included. User-defined functions cannot be used.

condition_expression includes one variable. The at sign (**@**) precedes each local variable. The expression refers to the value entered with the UPDATE or INSERT statement. Any name or symbol can be used to represent the value when creating the rule, but the first character must be the at sign (**@**).

Note

Avoid creating rules on expressions that use alias data types. Although rules can be created on expressions that use alias data types, after binding the rules to columns or alias data types, the expressions fail to compile when referenced.

Remarks

CREATE RULE cannot be combined with other Transact-SQL statements in a single batch. Rules do not apply to data already existing in the database at the time the rules are created, and rules cannot be bound to system data types.

A rule can be created only in the current database. After you create a rule, execute **sp_bindrule** to bind the rule to a column or to alias data type. A rule must be compatible with the column data type. For example, "@value LIKE A%" cannot be used as a rule for a numeric column. A rule cannot be bound to a **text**, **ntext**, **image**, **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, **xml**, CLR user-defined type, or **timestamp** column. A rule cannot be bound to a computed column.

Enclose character and date constants with single quotation marks (') and precede binary constants with 0x. If the rule is not compatible with the column to which it is bound, the SQL Server Database Engine returns an error message when a value is inserted, but not when the rule is bound.

A rule bound to an alias data type is activated only when you try to insert a value into, or to update, a database column of the alias data type. Because rules do not test variables, do not assign a value to an alias data type variable that would be rejected by a rule that is bound to a column of the same data type.

To get a report on a rule, use **sp_help**. To display the text of a rule, execute **sp_helptext** with the rule name as the parameter. To rename a rule, use **sp_rename**.

A rule must be dropped by using DROP RULE before a new one with the same name is created, and the rule must be unbound byusing **sp_unbindrule** before it is dropped. To unbind a rule from a column, use **sp_unbindrule**.

You can bind a new rule to a column or data type without unbinding the previous one; the new rule overrides the previous one. Rules bound to columns always take precedence over rules bound to alias data types. Binding a rule to a column replaces a rule already bound to the alias data type of that column. But binding a rule to a data type does not replace a rule bound to a column of that alias data type. The following table shows the precedence in effect when rules are bound to columns and to alias data types on which rules already exist.

New rule bound to	Old rule bound to alias data type	Old rule bound to Column
Alias data type	Old rule replaced	No change
Column	Old rule replaced	Old rule replaced

If a column has both a default and a rule associated with it, the default must fall within the domain defined by the rule. A default that conflicts with a rule is never inserted. The SQL Server Database Engine generates an error message each time it tries to insert such a default.

Permissions

To execute CREATE RULE, at a minimum, a user must have CREATE RULE permission in the current database and ALTER permission on the schema in which the rule is being created.

Examples

A. Creating a rule with a range

The following example creates a rule that restricts the range of integers inserted into the column or columns to which this rule is bound.

```
CREATE RULE range_rule
AS
@range>= $1000 AND @range <$20000;
```

B. Creating a rule with a list

The following example creates a rule that restricts the actual values entered into the column or columns (to which this rule is bound) to only those listed in the rule.

```
CREATE RULE list_rule
AS
@list IN ('1389', '0736', '0877');
```

C. Creating a rule with a pattern

The following example creates a rule to follow a pattern of any two characters followed by a hyphen (-), any number of characters or no characters, and ending with an integer from 0 through 9.

```
CREATE RULE pattern_rule
AS
@value LIKE '__-%[0-9]'
```

See Also

ALTER TABLE
CREATE DEFAULT
CREATE TABLE
DROP DEFAULT
DROP RULE
xpressions
sp_bindrule
sp_help
sp_helptext
sp_rename
sp_unbindrule
WHERE

CREATE SCHEMA

Creates a schema in the current database. The CREATE SCHEMA transaction can also create tables and views within the new schema, and set GRANT, DENY, or REVOKE permissions on those objects.

Transact-SQL Syntax Conventions

Syntax

CREATE SCHEMA **schema_name_clause** [<schema_element> [...**n**]]

<schema_name_clause> ::=

{
schema_name
| AUTHORIZATION **owner_name**
| schema_name AUTHORIZATION **owner_name**
}

<schema_element> ::=

```
{
   table_definition | view_definition | grant_statement |
   revoke_statement | deny_statement
}
```

Arguments

schema_name

Is the name by which the schema is identified within the database.

AUTHORIZATION owner_name

Specifies the name of the database-level principal that will own the schema. This principal may own other schemas, and may not use the current schema as its default schema.

table_definition

Specifies a CREATE TABLE statement that creates a table within the schema. The principal executing this statement must have CREATE TABLE permission on the current database.

view_definition

Specifies a CREATE VIEW statement that creates a view within the schema. The principal executing this statement must have CREATE VIEW permission on the current database.

grant_statement

Specifies a GRANT statement that grants permissions on any securable except the new schema.

revoke_statement

Specifies a REVOKE statement that revokes permissions on any securable except the new schema.

deny_statement

Specifies a DENY statement that denies permissions on any securable except the new schema.

Remarks

Note

Statements that contain CREATE SCHEMA AUTHORIZATION but do not specify a name are permitted for backward compatibility only.

CREATE SCHEMA can create a schema, the tables and views it contains, and GRANT, REVOKE, or DENY permissions on any securable in a single statement. This statement must be executed as a separate batch. Objects created by the CREATE SCHEMA statement are created inside the schema that is being created.

CREATE SCHEMA transactions are atomic. If any error occurs during the execution of a CREATE SCHEMA statement, none of the specified securables are created and no permissions are granted.

Securables to be created by CREATE SCHEMA can be listed in any order, except for views that reference other views. In that case, the referenced view must be created before the view that references it.

Therefore, a GRANT statement can grant permission on an object before the object itself is created, or a CREATE VIEW statement can appear before the CREATE TABLE statements that create the tables referenced by the view. Also, CREATE TABLE statements can declare foreign keys to tables that are defined later in the CREATE SCHEMA statement.

Note

DENY and REVOKE are supported inside CREATE SCHEMA statements. DENY and REVOKE clauses will be executed in the order in which they appear in the CREATE SCHEMA statement.

The principal that executes CREATE SCHEMA can specify another database principal as the owner of the schema being created. This requires additional permissions, as described in the "Permissions" section later in this topic.

The new schema is owned by one of the following database-level principals: database user, database role, or application role. Objects created within a schema are owned by the owner of the schema, and have a NULL **principal_id** in **sys.objects**. Ownership of schema-contained objects can be transferred to any database-level principal, but the schema owner always retains CONTROL permission on objects within the schema.

Caution

Beginning with SQL Server 2005, the behavior of schemas changed. As a result, code that assumes that schemas are equivalent to database users may no longer return correct results. Old catalog views, including , should not be used in a database in which any of the following DDL statements have ever been used: CREATE SCHEMA, ALTER SCHEMA, DROP SCHEMA, CREATE USER, ALTER USER, DROP USER, CREATE ROLE, ALTER ROLE, DROP ROLE, CREATE APPROLE, ALTER APPROLE, DROP APPROLE, ALTER AUTHORIZATION. In such databases you must instead use the new catalog views. The new catalog views take into account the separation of principals and schemas that was introduced in SQL Server 2005. For more information about catalog views, see .

When creating a database object, if you specify a valid domain principal (user or group) as the object owner, the domain principal will be added to the database as a schema. The new schema will be owned by that domain principal.

Deprecation Notice

CREATE SCHEMA statements that do not specify a schema name are currently supported for backward compatibility. Such statements do not actually create a schema inside the database, but they do create tables and views, and grant permissions. Principals do not need CREATE

SCHEMA permission to execute this earlier form of CREATE SCHEMA, because no schema is being created. This functionality will be removed from a future release of SQL Server.

Permissions

Requires CREATE SCHEMA permission on the database.

To create an object specified within the CREATE SCHEMA statement, the user must have the corresponding CREATE permission.

To specify another user as the owner of the schema being created, the caller must have IMPERSONATE permission on that user. If a database role is specified as the owner, the caller must have one of the following: membership in the role or ALTER permission on the role.

Note
For the backward-compatible syntax, no permissions to CREATE SCHEMA are checked because no schema is being created.

Examples

The following example creates schema `Sprockets` owned by `Annik` that contains table `NineProngs`. The statement grants `SELECT` to `Mandar` and denies `SELECT` to `Prasanna`. Note that `Sprockets` and `NineProngs` are created in a single statement.

```
USE AdventureWorks2012;
GO
CREATE SCHEMA Sprockets AUTHORIZATION Annik
    CREATE TABLE NineProngs (source int, cost int, partnumber int)
    GRANT SELECT ON SCHEMA::Sprockets TO Mandar
    DENY SELECT ON SCHEMA::Sprockets TO Prasanna;
GO
```

See Also

sys.schemas (Transact-SQL)
DROP SCHEMA (Transact-SQL)
GRANT (Transact-SQL)
DENY (Transact-SQL)
REVOKE (Transact-SQL)
CREATE VIEW (Transact-SQL)
EVENTDATA (Transact-SQL)
sys.schemas (Transact-SQL)
How to: Create a Database Schema

CREATE SEARCH PROPERTY LIST

Creates a new search property list. A search property list is used to specify one or more search properties that you want to include in a full-text index.

Important
CREATE SEARCH PROPERTY LIST, ALTER SEARCH PROPERTY LIST, and DROP SEARCH PROPERTY LIST are supported only under compatibility level 110. Under lower compatibility levels, these statements are not supported.

Transact-SQL Syntax Conventions

Syntax

```
CREATE SEARCH PROPERTY LIST new_list_name
  [ FROM [ database_name. ] source_list_name ]
  [ AUTHORIZATION owner_name ]
;
```

Arguments

new_list_name

Is the name of the new search property list. new_list_name is an identifier with a maximum of 128 characters. new_list_name must be unique among all property lists in the current database, and conform to the rules for identifiers. new_list_name will be used when the full-text index is created.

database_name

Is the name of the database where the property list specified by source_list_name is located. If not specified, database_name defaults to the current database.

database_name must specify the name of an existing database. The login for the current connection must be associated with an existing user ID in the database specified by database_name. You must also have the required permissions on the database.

source_list_name

Specifies that the new property list is created by copying an existing property list from database_name. If source_list_name does not exist, CREATE SEARCH PROPERTY LIST fails with an error. The search properties in source_list_name are inherited by new_list_name.

AUTHORIZATION owner_name

Specifies the name of a user or role to own of the property list. owner_name must either be the name of a role of which the current user is a member, or the current user must have IMPERSONATE permission on owner_name. If not specified, ownership is given to the current user.

Note
The owner can be changed by using the ALTER AUTHORIZATION Transact-SQL statement.

Remarks

Note
For information about property lists in general, see Search Document Properties with Search Property Lists.

By default, a new search property list is empty and you must alter it to manually to add one or more search properties. Alternatively, you can copy an existing search property list. In this case, the new list inherits the search properties of its source, but you can alter the new list to add or remove search properties. Any properties in the search property list at the time of the next full population are included in the full-text index.

A CREATE SEARCH PROPERTY LIST statement fails under any of the following conditions:

- If the database specified by database_name does not exist.
- If the list specified by source_list_name does not exist.
- If you do not have the correct permissions.

To add or remove properties from a list

- ALTER SEARCH PROPERTY LIST (Transact-SQL)
- **To drop a property list**
- DROP SEARCH PROPERTY LIST (Transact-SQL)

Permissions

Requires CREATE FULLTEXT CATALOG permissions in the current database and REFERENCES permissions on any database from which you copy a source property list.

Note
REFERENCES permission is required to associate the list with a full-text index. CONTROL permission is required to add and remove properties or drop the list. The property list owner can grant REFERENCES or CONTROL permissions on the list. Users with CONTROL permission can also grant REFERENCES permission to other users.

Examples

A. Creating an empty property list and associating it with an index

The following example creates a new search property list named `DocumentPropertyList`. The example then uses an ALTER FULLTEXT INDEX statement to associate the new property list with the full-text index of the `Production.Document` table in the `AdventureWorks` database, without starting a population.

Note

For an example that adds several predefined, well-known search properties to this search property list, see ALTER SEARCH PROPERTY LIST. After adding search properties to the list, the database administrator would need to use another ALTER FULLTEXT INDEX statement with the START FULL POPULATION clause.

```
CREATE SEARCH PROPERTY LIST DocumentPropertyList;
GO
USE AdventureWorks;
ALTER FULLTEXT INDEX ON Production.Document
   SET SEARCH PROPERTY LIST DocumentPropertyList
   WITH NO POPULATION;
GO
```

B. Creating a property list from an existing one

The following example creates a new the search property list, `JobCandidateProperties`, from the list created by Example A, `DocumentPropertyList`, which is associated with a full-text index in the `AdventureWorks` database. The example then uses an ALTER FULLTEXT INDEX statement to associate the new property list with the full-text index of the `HumanResources.JobCandidate` table in the `AdventureWorks` database. This ALTER FULLTEXT INDEX statement starts a full population, which is the default behavior of the SET SEARCH PROPERTY LIST clause.

```
CREATE SEARCH PROPERTY LIST JobCandidateProperties FROM
AdventureWorks.DocumentPropertyList;
GO
ALTER FULLTEXT INDEX ON HumanResources.JobCandidate
   SET SEARCH PROPERTY LIST JobCandidateProperties;
GO
```

See Also

ALTER SEARCH PROPERTY LIST (Transact-SQL)
DROP SEARCH PROPERTY LIST (Transact-SQL)
sys.registered_search_properties (Transact-SQL)
sys.registered_search_property_lists (Transact-SQL)
sys.dm_fts_index_keywords_by_property (Transact-SQL)
Using Search Property Lists to Search for Properties (Full-Text Search)
Obtaining a Property Set GUID and Property Integer Identifier for a Search Property List (SQL Server)

CREATE SEQUENCE

Creates a sequence object and specifies its properties. A sequence is a user-defined schema bound object that generates a sequence of numeric values according to the specification with which the sequence was created. The sequence of numeric values is generated in an ascending or descending order at a defined interval and can be configured to restart (cycle) when exhausted. Sequences, unlike identity columns, are not associated with specific tables. Applications refer to a sequence object to retrieve its next value. The relationship between sequences and tables is controlled by the application. User applications can reference a sequence object and coordinate the values across multiple rows and tables.

Unlike identity columns values that are generated when rows are inserted, an application can obtain the next sequence number without inserting the row by calling the NEXT VALUE FOR function. Use sp_sequence_get_range to get multiple sequence numbers at once.

For information and scenarios that use both **CREATE SEQUENCE** and the **NEXT VALUE FOR** function, see Creating and Using Sequence Numbers.

 Transact-SQL Syntax Conventions

Syntax

```
CREATE SEQUENCE [schema_name . ] sequence_name
    [ AS [ built_in_integer_type | user-defined_integer_type ] ]
    [ START WITH <constant> ]
    [ INCREMENT BY <constant> ]
    [ { MINVALUE [ <constant> ] } | { NO MINVALUE } ]
    [ { MAXVALUE [ <constant> ] } | { NO MAXVALUE } ]
    [ CYCLE | { NO CYCLE } ]
    [ { CACHE [ <constant> ] } | { NO CACHE } ]
    [ ; ]
```

Arguments

sequence_name

Specifies the unique name by which the sequence is known in the database. Type is **sysname**.

[built_in_integer_type | user-defined_integer_type

A sequence can be defined as any integer type. The following types are allowed.

- **tinyint** - Range 0 to 255
- **smallint** - Range -32,768 to 32,767
- **int** - Range -2,147,483,648 to 2,147,483,647
- **bigint** - Range -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
- **decimal** and **numeric** with a scale of 0.

- Any user-defined data type (alias type) that is based on one of the allowed types.

If no data type is provided, the **bigint** data type is used as the default.

START WITH <constant>

The first value returned by the sequence object. The **START** value must be a value less than or equal to the maximum and greater than or equal to the minimum value of the sequence object. The default start value for a new sequence object is the minimum value for an ascending sequence object and the maximum value for a descending sequence object.

INCREMENT BY <constant>

Value used to increment (or decrement if negative) the value of the sequence object for each call to the **NEXT VALUE FOR** function. If the increment is a negative value, the sequence object is descending; otherwise, it is ascending. The increment cannot be 0. The default increment for a new sequence object is 1.

[MINVALUE <constant> | NO MINVALUE]

Specifies the bounds for the sequence object. The default minimum value for a new sequence object is the minimum value of the data type of the sequence object. This is zero for the **tinyint** data type and a negative number for all other data types.

[MAXVALUE <constant> | NO MAXVALUE

Specifies the bounds for the sequence object. The default maximum value for a new sequence object is the maximum value of the data type of the sequence object.

[CYCLE | NO CYCLE]

Property that specifies whether the sequence object should restart from the minimum value (or maximum for descending sequence objects) or throw an exception when its minimum or maximum value is exceeded. The default cycle option for new sequence objects is NO CYCLE.

Note that cycling restarts from the minimum or maximum value, not from the start value.

[CACHE [<constant>] | NO CACHE]

Increases performance for applications that use sequence objects by minimizing the number of disk IOs that are required to generate sequence numbers.

For example, if a cache size of 50 is chosen, SQL Server does not keep 50 individual values cached. It only caches the current value and the number of values left in the cache. This means that the amount of memory required to store the cache is always two instances of the data type of the sequence object.

Note

If the cache option is enabled without specifying a cache size, the Database Engine will select a size. However, users should not rely upon the selection being consistent. Microsoft might change the method of calculating the cache size without notice.

When created with the **CACHE** option, an unexpected shutdown (such as a power failure) may result in the loss of sequence numbers remaining in the cache.

General Remarks

Sequence numbers are generated outside the scope of the current transaction. They are consumed whether the transaction using the sequence number is committed or rolled back.

Cache management

To improve performance, SQL Server pre-allocates the number of sequence numbers specified by the **CACHE** argument.

For an example, a new sequence is created with a starting value of 1 and a cache size of 15. When the first value is needed, values 1 through 15 are made available from memory. The last cached value (15) is written to the system tables on the disk. When all 15 numbers are used, the next request (for number 16) will cause the cache to be allocated again. The new last cached value (30) will be written to the system tables.

If the Database Engine is stopped after you use 22 numbers, the next intended sequence number in memory (23) is written to the system tables, replacing the previously stored number.

After SQL Server restarts and a sequence number is needed, the starting number is read from the system tables (23). The cache amount of 15 numbers (23-38) is allocated to memory and the next non-cache number (39) is written to the system tables.

If the Database Engine stops abnormally for an event such as a power failure, the sequence restarts with the number read from system tables (39). Any sequence numbers allocated to memory (but never requested by a user or application) are lost. This functionality may leave gaps, but guarantees that the same value will never be issued two times for a single sequence object unless it is defined as **CYCLE** or is manually restarted.

The cache is maintained in memory by tracking the current value (the last value issued) and the number of values left in the cache. Therefore, the amount of memory used by the cache is always two instances of the data type of the sequence object.

Setting the cache argument to **NO CACHE** writes the current sequence value to the system tables every time that a sequence is used. This might slow performance by increasing disk access, but reduces the chance of unintended gaps. Gaps can still occur if numbers are requested using the **NEXT VALUE FOR** or **sp_sequence_get_range** functions, but then the numbers are either not used or are used in uncommitted transactions.

When a sequence object uses the **CACHE** option, if you restart the sequence object, or alter the **INCREMENT**, **CYCLE**, **MINVALUE**, **MAXVALUE**, or the cache size properties, it will cause the cache to be written to the system tables before the change occurs. Then the cache is reloaded starting with the current value (i.e. no numbers are skipped). Changing the cache size takes effect immediately.

CACHE option when cached values are available

The following process occurs every time that a sequence object is requested to generate the next value for the **CACHE** option if there are unused values available in the in-memory cache for the sequence object.

1. The next value for the sequence object is calculated.

2. The new current value for the sequence object is updated in memory.
3. The calculated value is returned to the calling statement.

CACHE option when the cache is exhausted

The following process occurs every time a sequence object is requested to generate the next value for the **CACHE** option if the cache has been exhausted:

1. The next value for the sequence object is calculated.
2. The last value for the new cache is calculated.
3. The system table row for the sequence object is locked, and the value calculated in step 2 (the last value) is written to the system table. A cache-exhausted xevent is fired to notify the user of the new persisted value.

NO CACHE option

The following process occurs every time that a sequence object is requested to generate the next value for the **NO CACHE** option:

1. The next value for the sequence object is calculated.
2. The new current value for the sequence object is written to the system table.
3. The calculated value is returned to the calling statement.

Metadata

For information about sequences, query sys.sequences.

Security

Permissions

Requires **CREATE SEQUENCE**, **ALTER**, or **CONTROL** permission on the SCHEMA.

- Members of the db_owner and db_ddladmin fixed database roles can create, alter, and drop sequence objects.
- Members of the db_owner and db_datawriter fixed database roles can update sequence objects by causing them to generate numbers.

The following example grants the user AdventureWorks\Larry permission to create sequences in the Test schema.

```
GRANT CREATE SEQUENCE ON SCHEMA::Test TO [AdventureWorks\Larry]
```

Ownership of a sequence object can be transferred by using the **ALTER AUTHORIZATION** statement.

If a sequence uses a user-defined data type, the creator of the sequence must have REFERENCES permission on the type.

Audit

To audit **CREATE SEQUENCE**, monitor the **SCHEMA_OBJECT_CHANGE_GROUP**.

Examples

For examples of creating sequences and using the **NEXT VALUE FOR** function to generate sequence numbers, see Creating and Using Sequence Numbers.

Most of the following examples create sequence objects in a schema named Test.

To create the Test schema, execute the following statement.

```
-- CREATE SCHEMA Test ;
GO
```

A. Creating a sequence that increases by 1

In the following example, Thierry creates a sequence named CountBy1 that increases by one every time that it is used.

```
CREATE SEQUENCE Test.CountBy1
    START WITH 1
    INCREMENT BY 1 ;
GO
```

B. Creating a sequence that decreases by 1

The following example starts at 0 and counts into negative numbers by one every time it is used.

```
CREATE SEQUENCE Test.CountByNeg1
    START WITH 0
    INCREMENT BY -1 ;
GO
```

C. Creating a sequence that increases by 5

The following example creates a sequence that increases by 5 every time it is used.

```
CREATE SEQUENCE Test.CountBy1
    START WITH 5
    INCREMENT BY 5 ;
GO
```

D. Creating a sequence that starts with a designated number

After importing a table, Thierry notices that the highest ID number used is 24,328. Thierry needs a sequence that will generate numbers starting at 24,329. The following code creates a sequence that starts with 24,329 and increments by 1.

```
CREATE SEQUENCE Test.ID_Seq
    START WITH 24329
    INCREMENT BY 1 ;
GO
```

E. Creating a sequence using default values

The following example creates a sequence using the default values.

```
CREATE SEQUENCE Test.TestSequence ;
```

Execute the following statement to view the properties of the sequence.

```
SELECT * FROM sys.sequences WHERE name = 'TestSequence' ;
```

A partial list of the output demonstrates the default values.

start_value	-9223372036854775808
increment	1
mimimum_value	-9223372036854775808
maximum_value	9223372036854775807
is_cycling	0
is_cached	1
current_value	-9223372036854775808

F. Creating a sequence with a specific data type

The following example creates a sequence using the **smallint** data type, with a range from -32,768 to 32,767.

```
CREATE SEQUENCE SmallSeq
    AS smallint ;
```

G. Creating a sequence using all arguments

The following example creates a sequence named DecSeq using the **decimal** data type, having a range from 0 to 255. The sequence starts with 125 and increments by 25 every time that a number is generated. Because the sequence is configured to cycle when the value exceeds the maximum value of 200, the sequence restarts at the minimum value of 100.

```
CREATE SEQUENCE Test.DecSeq
    AS decimal(3,0)
    START WITH 125
    INCREMENT BY 25
    MINVALUE 100
    MAXVALUE 200
    CYCLE
    CACHE 3
;
```

Execute the following statement to see the first value; the `START WITH` option of 125.

```
SELECT NEXT VALUE FOR Test.DecSeq;
```

Execute the statement three more times to return 150, 175, and 200.

Execute the statement again to see how the start value cycles back to the `MINVALUE` option of 100.

Execute the following code to confirm the cache size and see the current value.

```
SELECT cache_size, current_value
FROM sys.sequences
WHERE name = 'DecSeq' ;
```

See Also

ALTER SEQUENCE (Transact-SQL)

DROP SEQUENCE (Transact-SQL)

NEXT VALUE FOR function (Transact-SQL)

Creating and Using Sequence Numbers

CREATE SERVER AUDIT

Creates a server audit object using SQL Server Audit. For more information, see Understanding SQL Server Audit.

 Transact-SQL Syntax Conventions

Syntax

```
CREATE SERVER AUDIT audit_name
{
   TO { [ FILE (<file_options> [ , ...n ] ) ] | APPLICATION_LOG | SECURITY_LOG }
   [ WITH ( <audit_options> [ , ...n ] ) ]
   [ WHERE <predicate_expression> ]
}
[ ; ]

<file_options>::=
{
      FILEPATH = 'os_file_path'
   [ , MAXSIZE = { max_size { MB | GB | TB } | UNLIMITED } ]
   [ , { MAX_ROLLOVER_FILES = { integer | UNLIMITED } } | { MAX_FILES = integer } ]
```

[, RESERVE_DISK_SPACE = { ON | OFF }]
}

<audit_options>::=

{
[QUEUE_DELAY = **integer**]
[, ON_FAILURE = { CONTINUE | SHUTDOWN | FAIL_OPERATION }]
[, AUDIT_GUID = **uniqueidentifier**]
}

<predicate_expression>::=

{
[**NOT**] <predicate_factor>
[{ **AND** | **OR** } [**NOT**] { <predicate_factor> }]
[,...n]
}

<predicate_factor>::=

event_field_name { = | **<** **>** | **!** = | **>** | **>** = | **<** | **<** = } { **number** | **'** **string** **'** }

Arguments

TO { FILE | APPLICATION_LOG | SECURITY_LOG }

Determines the location of the audit target. The options are a binary file, The Windows Application log, or the Windows Security log. SQL Server cannot write to the Windows Security log without configuring additional settings in Windows. For more information, see How to: Write Server Audit Events to the Security Log.

FILEPATH = 'os_file_path'

The path of the audit log. The file name is generated based on the audit name and audit GUID.

MAXSIZE = { max_size }

Specifies the maximum size to which the audit file can grow. The max_size value must be an integer followed by MB, GB, TB, or UNLIMITED. The minimum size that you can specify for max_size is 2 MB and the maximum is 2,147,483,647 TB. When UNLIMITED is specified, the file grows until the disk is full. Specifying a value lower than 2 MB will raise the error MSG_MAXSIZE_TOO_SMALL. The default value is UNLIMITED.

MAX_ROLLOVER_FILES = { integer | UNLIMITED }

Specifies the maximum number of files to retain in the file system in addition to the current file. The MAX_ROLLOVER_FILES value must be an integer or UNLIMITED. The default value is UNLIMITED. This parameter is evaluated whenever the audit restarts (which can happen when the instance of the Database Engine restarts or when the audit is turned off and then on again) or when a new file is needed because the MAXSIZE has been reached. When MAX_ROLLOVER_FILES is evaluated, if the number of files exceeds the MAX_ROLLOVER_FILES setting, the oldest file is deleted. As a result, when the setting of MAX_ROLLOVER_FILES is 0 a new file is created each time the MAX_ROLLOVER_FILES setting is evaluated. Only one file is automatically deleted when MAX_ROLLOVER_FILES setting is evaluated, so when the value of MAX_ROLLOVER_FILES is decreased, the number of files will not shrink unless old files are manually deleted. The maximum number of files that can be specified is 2,147,483,647.

MAX_FILES = integer

Specifies the maximum number of audit files that can be created. Does not rollover to the first file when the limit is reached. When the MAX_FILES limit is reached, any action that causes additional audit events to be generated will fail with an error.

RESERVE_DISK_SPACE = { ON | OFF }

This option pre-allocates the file on the disk to the MAXSIZE value. It applies only if MAXSIZE is not equal to UNLIMITED. The default value is OFF.

QUEUE_DELAY = integer

Determines the time, in milliseconds, that can elapse before audit actions are forced to be processed. A value of 0 indicates synchronous delivery. The minimum settable query delay value is 1000 (1 second), which is the default. The maximum is 2,147,483,647 (2,147,483.647 seconds or 24 days, 20 hours, 31 minutes, 23.647 seconds). Specifying an invalid number will raise the error MSG_INVALID_QUEUE_DELAY.

ON_FAILURE = { CONTINUE | SHUTDOWN | FAIL_OPERATION }

Indicates whether the instance writing to the target should fail, continue, or stop SQL Server if the target cannot write to the audit log. The default value is CONTINUE.

CONTINUE

SQL Server operations continue. Audit records are not retained. The audit continues to attempt to log events and will resume if the failure condition is resolved. Selecting the continue option can allow unaudited activity which could violate your security policies. Use this option, when continuing operation of the Database Engine is more important than maintaining a complete audit.

SHUTDOWN

Forces a server shut down when the server instance writing to the target cannot write data to the audit target. The login issuing this must have the **SHUTDOWN** permission. If the logon does not have this permission, this function will fail and an error message will be

raised. No audited events occur. Use the option when an audit failure could compromise the security or integrity of the system.

FAIL_OPERATION

Database actions fail if they cause audited events. Actions which do not cause audited events can continue, but no audited events can occur. The audit continues to attempt to log events and will resume if the failure condition is resolved. Use this option when maintaining a complete audit is more important than full access to the Database Engine.

AUDIT_GUID = uniqueidentifier

To support scenarios such as database mirroring, an audit needs a specific GUID that matches the GUID found in the mirrored database. The GUID cannot be modified after the audit has been created.

predicate_expression

Specifies the predicate expression used to determine if an event should be processed or not. Predicate expressions are limited to 3000 characters, which limits string arguments.

event_field_name

Is the name of the event field that identifies the predicate source. Audit fields are described in fn_get_audit_file (Transact-SQL). All fields can be audited except `file_name` and `audit_file_offset`.

number

Is any numeric type including **decimal**. Limitations are the lack of available physical memory or a number that is too large to be represented as a 64-bit integer.

' string '

Either an ANSI or Unicode string as required by the predicate compare. No implicit string type conversion is performed for the predicate compare functions. Passing the wrong type results in an error.

Remarks

When a server audit is created, it is in a disabled state.

The CREATE SERVER AUDIT statement is in a transaction's scope. If the transaction is rolled back, the statement is also rolled back.

Permissions

To create, alter, or drop a server audit, principals require the ALTER ANY SERVER AUDIT or the CONTROL SERVER permission.

When you are saving audit information to a file, to help prevent tampering, restrict access to the file location.

Examples

A. Creating a server audit with a file target

The following example creates a server audit called `HIPPA_Audit` with a binary file as the target and no options.

```
CREATE SERVER AUDIT HIPAA_Audit
    TO FILE ( FILEPATH ='\\SQLPROD_1\Audit\' );
```

B. Creating a server audit with a Windows Application log target with options

The following example creates a server audit called `HIPPA_Audit` with the target set for the Windows Application log. The queue is written every second and shuts down the SQL Server engine on failure.

```
CREATE SERVER AUDIT HIPAA_Audit
    TO APPLICATION_LOG
    WITH ( QUEUE_DELAY = 1000,  ON_FAILURE = SHUTDOWN);
```

C. Creating a server audit containing a WHERE clause

The following example creates a database, schema, and two tables for the example. The table named `DataSchema.SensitiveData` will contain confidential data and access to the table must be recorded in the audit. The table named `DataSchema.GeneralData` does not contain confidential data. The database audit specification audits access to all objects in the `DataSchema` schema. The server audit is created with a WHERE clause that limits the server audit to only the `SensitiveData` table. The server audit presumes a audit folder exists at `C:\SQLAudit`.

```
CREATE DATABASE TestDB;
GO
USE TestDB;
GO
CREATE SCHEMA DataSchema;
GO
CREATE TABLE DataSchema.GeneralData (ID int PRIMARY KEY, DataField
varchar(50) NOT NULL);
GO
CREATE TABLE DataSchema.SensitiveData (ID int PRIMARY KEY, DataField
varchar(50) NOT NULL);
GO
-- Create the server audit in the master database
USE master;
GO
CREATE SERVER AUDIT AuditDataAccess
    TO FILE ( FILEPATH ='C:\SQLAudit\' )
```

```
    WHERE object_name = 'SensitiveData' ;
GO
ALTER SERVER AUDIT AuditDataAccess WITH (STATE = ON);
GO
-- Create the database audit specification in the TestDB database
USE TestDB;
GO
CREATE DATABASE AUDIT SPECIFICATION [FilterForSensitiveData]
FOR SERVER AUDIT [AuditDataAccess]
ADD (SELECT ON SCHEMA::[DataSchema] BY [public])
WITH (STATE = ON);
GO
-- Trigger the audit event by selecting from tables
SELECT ID, DataField FROM DataSchema.GeneralData;
SELECT ID, DataField FROM DataSchema.SensitiveData;
GO
-- Check the audit for the filtered content
SELECT * FROM
fn_get_audit_file('C:\SQLAudit\AuditDataAccess_*.sqlaudit',default,default);
GO
```

See Also

ALTER SERVER AUDIT (Transact-SQL)
DROP SERVER AUDIT (Transact-SQL)
CREATE SERVER AUDIT SPECIFICATION (Transact-SQL)
ALTER SERVER AUDIT SPECIFICATION (Transact-SQL)
DROP SERVER AUDIT SPECIFICATION (Transact-SQL)
CREATE DATABASE AUDIT SPECIFICATION (Transact-SQL)
ALTER DATABASE AUDIT SPECIFICATION (Transact-SQL)
DROP DATABASE AUDIT SPECIFICATION (Transact-SQL)
ALTER AUTHORIZATION (Transact-SQL)
fn_get_audit_file (Transact-SQL)
sys.server_audits (Transact-SQL)
sys.server_file_audits (Transact-SQL)
sys.server_audit_specifications (Transact-SQL)

sys.server_audit_specifications_details (Transact-SQL)
sys.database_audit_specifications (Transact-SQL)
sys.database_audit_specification_details (Transact-SQL)
sys.dm_server_audit_status
sys.dm_audit_actions
sys.dm_audit_class_type_map
Create a Server Audit and Server Audit Specification

CREATE SERVER AUDIT SPECIFICATION

Creates a server audit specification object using the SQL Server Audit feature. For more information, see Understanding SQL Server Audit.

 Transact-SQL Syntax Conventions

Syntax

```
CREATE SERVER AUDIT SPECIFICATION audit_specification_name
FOR SERVER AUDIT audit_name
{
   { ADD ( { audit_action_group_name } )
      } [ , ...n]
   [ WITH ( STATE = { ON | OFF } ) ]
}
[ ; ]
```

Arguments

audit_specification_name

Name of the server audit specification.

audit_name

Name of the audit to which this specification is applied.

audit_action_group_name

Name of a group of server-level auditable actions. For a list of Audit Action Groups, see SQL Server Audit Action Groups and Actions.

WITH (STATE = { ON | OFF })

Enables or disables the audit from collecting records for this audit specification.

Remarks

An audit must exist before creating a server audit specification for it. When a server audit specification is created, it is in a disabled state.

Permissions

Users with the ALTER ANY SERVER AUDIT permission can create server audit specifications and bind them to any audit.

After a server audit specification is created, it can be viewed by principals with the, CONTROL SERVER, or ALTER ANY SERVER AUDIT permissions, the sysadmin account, or principals having explicit access to the audit.

Examples

The following example creates a server audit specification called `HIPPA_Audit_Specification` that audits failed logins, for a SQL Server Audit called `HIPPA_Audit`.

```
CREATE SERVER AUDIT SPECIFICATION HIPPA_Audit_Specification
FOR SERVER AUDIT HIPPA_Audit
    ADD (FAILED_LOGIN_GROUP);
GO
```

For a full example about how to create an audit, see Understanding SQL Server Audit.

See Also

CREATE SERVER AUDIT (Transact-SQL)
ALTER SERVER AUDIT (Transact-SQL)
DROP SERVER AUDIT (Transact-SQL)
ALTER SERVER AUDIT SPECIFICATION (Transact-SQL)
DROP SERVER AUDIT SPECIFICATION (Transact-SQL)
CREATE DATABASE AUDIT SPECIFICATION (Transact-SQL)
ALTER DATABASE AUDIT SPECIFICATION (Transact-SQL)
DROP DATABASE AUDIT SPECIFICATION (Transact-SQL)
ALTER AUTHORIZATION (Transact-SQL)
fn_get_audit_file (Transact-SQL)
sys.server_audits (Transact-SQL)
sys.server_file_audits (Transact-SQL)
sys.server_audit_specifications (Transact-SQL)
sys.server_audit_specifications_details (Transact-SQL)
sys.database_ audit_specifications (Transact-SQL)
sys.audit_database_specification_details (Transact-SQL)
sys.dm_server_audit_status
sys.dm_audit_actions

Create a Server Audit and Server Audit Specification

CREATE SERVER ROLE

Creates a new user-defined server role.

 Transact-SQL Syntax Conventions

Syntax

```
CREATE SERVER ROLE role_name [ AUTHORIZATION server_principal ]
```

Arguments

role_name

Is the name of the server role to be created.

AUTHORIZATION server_principal

Is the login that will own the new server role. If no login is specified, the server role will be owned by the login that executes CREATE SERVER ROLE.

Remarks

Server roles are server-level securables. After you create a server role, configure the server-level permissions of the role by using GRANT, DENY, and REVOKE. To add logins to or remove logins from a server role, use ALTER SERVER ROLE. To drop a server role, use DROP SERVER ROLE. For more information, see sys.server_principals (Transact-SQL).

You can view the server roles by querying the sys.server_role_members and sys.server_principals catalog views.

Server roles cannot be granted permission on database-level securables. To create database roles, see CREATE ROLE (Transact-SQL).

Permissions

Requires CREATE SERVER ROLE permission or membership in the sysadmin fixed server role.

Also requires IMPERSONATE on the server_principal for logins, ALTER permission for server roles used as the server_principal, or membership in a Windows group that is used as the server_principal.

This will fire the Audit Server Principal Management event with the object type set to server role and event type to add.

When you use the AUTHORIZATION option to assign server role ownership, the following permissions are also required:

- To assign ownership of a server role to another login, requires IMPERSONATE permission on that login.

- To assign ownership of a server role to another server role, requires membership in the recipient server role or ALTER permission on that server role.

Examples

A. Creating a server role that is owned by a login

The following example creates the server role `buyers` that is owned by login `BenMiller`.

```
USE master;
CREATE SERVER ROLE buyers AUTHORIZATION BenMiller;
GO
```

B. Creating a server role that is owned by a fixed server role

The following example creates the server role `auditors` that is owned the `securityadmin` fixed server role.

```
USE master;
CREATE SERVER ROLE auditors AUTHORIZATION securityadmin;
GO
```

See Also

DROP SERVER ROLE (Transact-SQL)
Principals
EVENTDATA (Transact-SQL)
sp_addrolemember (Transact-SQL)
sys.database_role_members (Transact-SQL)
sys.database_principals (Transact-SQL)

CREATE SERVICE

Creates a new service. A Service Broker service is a name for a specific task or set of tasks. Service Broker uses the name of the service to route messages, deliver messages to the correct queue within a database, and enforce the contract for a conversation.

 Transact-SQL Syntax Conventions

Syntax

```
CREATE SERVICE service_name
  [ AUTHORIZATION owner_name ]
  ON QUEUE [ schema_name. ]queue_name
  [ ( contract_name | [DEFAULT] [ ,...n ] ) ]
[ ; ]
```

Arguments

service_name

Is the name of the service to create. A new service is created in the current database and owned by the principal specified in the AUTHORIZATION clause. Server, database, and schema names cannot be specified. The service_name must be a valid **sysname**.

Note

Do not create a service that uses the keyword ANY for the service_name. When you specify ANY for a service name in CREATE BROKER PRIORITY, the priority is considered for all services. It is not limited to a service whose name is ANY.

AUTHORIZATION owner_name

Sets the owner of the service to the specified database user or role. When the current user is **dbo** or **sa**, owner_name may be the name of any valid user or role. Otherwise, owner_name must be the name of the current user, the name of a user that the current user has IMPERSONATE permission for, or the name of a role to which the current user belongs.

ON QUEUE [schema_name .] queue_name

Specifies the queue that receives messages for the service. The queue must exist in the same database as the service. If no schema_name is provided, the schema is the default schema for the user that executes the statement.

contract_name

Specifies a contract for which this service may be a target. Service programs initiate conversations to this service using the contracts specified. If no contracts are specified, the service may only initiate conversations.

[DEFAULT]

Specifies that the service may be a target for conversations that follow the DEFAULT contract. In the context of this clause, DEFAULT is not a keyword, and must be delimited as an identifier. The DEFAULT contract allows both sides of the conversation to send messages of message type DEFAULT. Message type DEFAULT uses validation NONE.

Remarks

A service exposes the functionality provided by the contracts with which it is associated, so that they can be used by other services. The CREATE SERVICE statement specifies the contracts that this service is the target for. A service can only be a target for conversations that use the contracts specified by the service. A service that specifies no contracts exposes no functionality to other services.

Conversations initiated from this service may use any contract. You create a service without specifying contracts when the service only initiates conversations.

When Service Broker accepts a new conversation from a remote service, the name of the target service determines the queue where the broker places messages in the conversation.

Permissions

Permission for creating a service defaults to members of the **db_ddladmin** or **db_owner** fixed database roles and the **sysadmin** fixed server role. The user executing the CREATE SERVICE statement must have REFERENCES permission on the queue and all contracts specified.

REFERENCES permission for a service defaults to the owner of the service, members of the **db_ddladmin** or **db_owner** fixed database roles, and members of the **sysadmin** fixed server role. SEND permissions for a service default to the owner of the service, members of the **db_owner** fixed database role, and members of the **sysadmin** fixed server role.

A service may not be a temporary object. Service names beginning with **#** are allowed, but are permanent objects.

Examples

A. Creating a service with one contract

The following example creates the service `//Adventure-Works.com/Expenses` on the `ExpenseQueue` queue in the `dbo` schema. Dialogs that target this service must follow the contract `//Adventure-Works.com/Expenses/ExpenseSubmission`.

```
CREATE SERVICE [//Adventure-Works.com/Expenses]
    ON QUEUE [dbo].[ExpenseQueue]
    ([//Adventure-Works.com/Expenses/ExpenseSubmission]) ;
```

B. Creating a service with multiple contracts

The following example creates the service `//Adventure-Works.com/Expenses` on the `ExpenseQueue` queue. Dialogs that target this service must either follow the contract `//Adventure-Works.com/Expenses/ExpenseSubmission` or the contract `//Adventure-Works.com/Expenses/ExpenseProcessing`.

```
CREATE SERVICE [//Adventure-Works.com/Expenses] ON QUEUE ExpenseQueue
    ([//Adventure-Works.com/Expenses/ExpenseSubmission],
     [//Adventure-Works.com/Expenses/ExpenseProcessing]) ;
```

C. Creating a service with no contracts

The following example creates the service `//Adventure-Works.com/Expenses on the ExpenseQueue` queue. This service has no contract information. Therefore, the service can only be the initiator of a dialog.

```
CREATE SERVICE [//Adventure-Works.com/Expenses] ON QUEUE ExpenseQueue ;
```

See Also

EVENTDATA (Transact-SQL)

DROP SERVICE

EVENTDATA

CREATE SPATIAL INDEX

Creates a spatial index on a specified table and column. An index can be created before there is data in the table. Indexes can be created on tables or views in another database by specifying a qualified database name. Spatial indexes require the table to have a clustered primary key.

Note
For information about spatial indexes, see Spatial Indexes Overview.

Transact-SQL Syntax Conventions

Syntax

Create Spatial Index

```
CREATE SPATIAL INDEX index_name
  ON <object> ( spatial_column_name )
    {
      <geometry_tessellation> | <geography_tessellation>
    }
  [ ON { filegroup_name | "default" } ]
;

<object> ::=
    [ database_name. [ schema_name ] . | schema_name. ]
      table_name

<geometry_tessellation> ::=
{
  <geometry_automatic_grid_tessellation> | <geometry_manual_grid_tessellation>
}


<geometry_automatic_grid_tessellation> ::=
{
    [ USING GEOMETRY_AUTO_GRID ]
        WITH  (
       <bounding_box>
```

```
        [ [,] <tessellation_cells_per_object> [ ,...n] ]
        [ [,] <spatial_index_option> [ ,...n] ]
    )
}
```

<geometry_manual_grid_tessellation> ::=

```
{
    [ USING GEOMETRY_GRID ]
      WITH (
            <bounding_box>
              [ [,]<tessellation_grid> [ ,...n] ]
              [ [,]<tessellation_cells_per_object> [ ,...n] ]
              [ [,]<spatial_index_option> [ ,...n] ]
  )
}
```

<geography_tessellation> ::=

```
{
    <geography_automatic_grid_tessellation> | <geography_manual_grid_tessellation>
}
```

<geography_automatic_grid_tessellation> ::=

```
{
   [ USING GEOGRAPHY_AUTO_GRID ]
   [ WITH (
      [ [,] <tessellation_cells_per_object> [ ,...n] ]
      [ [,] <spatial_index_option> ]
    ) ]
}
```

<geography_manual_grid_tessellation> ::=

{

[USING GEOGRAPHY_GRID]

[WITH (

[<tessellation_grid> [,...n]]

[[,] <tessellation_cells_per_object> [,...n]]

[[,] <spatial_index_option> [,...n]]

)]

}

<bounding_box> ::=

{

BOUNDING_BOX = **(** {

xmin, ymin, xmax, ymax

| <named_bb_coordinate>, <named_bb_coordinate>, <named_bb_coordinate>, <named_bb_coordinate>

} **)**

}

<named_bb_coordinate> ::= { XMIN = **xmin** | YMIN = **ymin** | XMAX = **xmax** | YMAX=**ymax** }

<tesselation_grid> ::=

{

GRIDS = **(** { <grid_level> [,...**n**] | <grid_size>, <grid_size>, <grid_size>, <grid_size> }

)

}

<tesseallation_cells_per_object> ::=

{

CELLS_PER_OBJECT = **n**

}

<grid_level> ::=

{

LEVEL_1 = <grid_size>

| LEVEL_2 = <grid_size>

```
  | LEVEL_3 = <grid_size>
  | LEVEL_4 = <grid_size>
}

<grid_size> ::= { LOW | MEDIUM | HIGH }

<spatial_index_option> ::=
{
    PAD_INDEX = { ON | OFF }
  | FILLFACTOR = fillfactor
  | SORT_IN_TEMPDB = { ON | OFF }
  | IGNORE_DUP_KEY = OFF
  | STATISTICS_NORECOMPUTE = { ON | OFF }
  | DROP_EXISTING = { ON | OFF }
  | ONLINE =  OFF
  | ALLOW_ROW_LOCKS = { ON | OFF }
  | ALLOW_PAGE_LOCKS = { ON | OFF }
  | MAXDOP = max_degree_of_parallelism
    | DATA_COMPRESSION = { NONE | ROW | PAGE }
}
```

Arguments

index_name

Is the name of the index. Index names must be unique within a table but do not have to be unique within a database. Index names must follow the rules of identifiers.

ON <object> (spatial_column_name)

Specifies the object (database, schema, or table) on which the index is to be created and the name of spatial column.

spatial_column_name specifies the spatial column on which the index is based. Only one spatial column can be specified in a single spatial index definition; however, multiple spatial indexes can be created on a **geometry** or **geography** column.

USING

Indicates the tessellation scheme for the spatial index. This parameter uses the type-specific value, shown in the following table:

Data type of column	Tessellation scheme
geometry	GEOMETRY_GRID
geometry	GEOMETRY_AUTO_GRID
geography	GEOGRAPY_GRID
geography	GEOGRAPHY_AUTO_GRID

A spatial index can be created only on a column of type **geometry** or **geography**. Otherwise, an error is raised. Also, if an invalid parameter for a given type is passed, an error is raised.

nNote

For information about how SQL Server implements tessellation, see Spatial Indexes Overview.

ON filegroup_name

Creates the specified index on the specified filegroup. If no location is specified and the table is not partitioned, the index uses the same filegroup as the underlying table. The filegroup must already exist.

ON "default"

Creates the specified index on the default filegroup.

The term default, in this context, is not a keyword. It is an identifier for the default filegroup and must be delimited, as in ON "default" or ON [default]. If "default" is specified, the QUOTED_IDENTIFIER option must be ON for the current session. This is the default setting. For more information, see SET QUOTED_IDENTIFIER (Transact-SQL).

<object>::=

Is the fully qualified or non-fully qualified object to be indexed.

database_name

Is the name of the database.

schema_name

Is the name of the schema to which the table belongs.

table_name

Is the name of the table to be indexed.

USING Options

GEOMETRY_GRID

Specifies the **geometry** grid tessellation scheme that you are using. GEOMETRY_GRID can be specified only on a column of the **geometry** data type. GEOMETRY_GRID allows for manual adjusting of the tessellation scheme.

GEOMETRY_AUTO_GRID

Can be specified only on a column of the geometry data type. This is the default for this data type and does not need to be specified.

GEOGRAPHY_GRID

Specifies the geography grid tessellation scheme. GEOGRAPHY_GRID can be specified only on a column of the **geography** data type.

GEOGRAPHY_AUTO_GRID

Can be specified only on a column of the geography data type. This is the default for this data type and does not need to be specified.

WITH Options

BOUNDING_BOX

Specifies a numeric four-tuple that defines the four coordinates of the bounding box: the x-min and y-min coordinates of the lower-left corner, and the x-max and y-max coordinates of the upper-right corner.

xmin

Specifies the x-coordinate of the lower-left corner of the bounding box.

ymin

Specifies the y-coordinate of the lower-left corner of the bounding box.

xmax

Specifies the x-coordinate of the upper-right corner of the bounding box.

ymax

Specifies the y-coordinate of the upper-right corner of the bounding box.

XMIN = xmin

Specifies the property name and value for the x-coordinate of the lower-left corner of the bounding box.

YMIN = ymin

Specifies the property name and value for the y-coordinate of the lower-left corner of the bounding box.

XMAX = xmax

Specifies the property name and value for the x-coordinate of the upper-right corner of the bounding box.

YMAX = ymax

Specifies the property name and value for the y-coordinate of upper-right corner of the bounding box

Bounding-box coordinates apply only within a USING GEOMETRY_GRID clause.

xmax must be greater than xmin and ymax must be greater than ymin. You can specify any valid float value representation, assuming that: xmax > xmin and ymax > ymin. Otherwise the appropriate errors are raised.

There are no default values.

The bounding-box property names are case-insensitive regardless of the database collation.

To specify property names, you must specify each of them once and only once. You can specify them in any order. For example, the following clauses are equivalent:

- BOUNDING_BOX = (XMIN = xmin, YMIN = ymin, XMAX = xmax, YMAX = ymax)
- BOUNDING_BOX = (XMIN = xmin, XMAX = xmax, YMIN = ymin, YMAX = ymax)

GRIDS

Defines the density of the grid at each level of a tessellation scheme. When GEOMETRY_AUTO_GRID and GEOGRAPHY_AUTO_GRID are selected, this option is disabled.

Note

For information about tessellation, see Spatial Indexes Overview.

The GRIDS parameters are as follows:

LEVEL_1

Specifies the first-level (top) grid.

LEVEL_2

Specifies the second-level grid.

LEVEL_3

Specifies the third-level grid.

LEVEL_4

Specifies the fourth-level grid.

LOW

Specifies the lowest possible density for the grid at a given level. LOW equates to 16 cells (a 4x4 grid).

MEDIUM

Specifies the medium density for the grid at a given level. MEDIUM equates to 64 cells (an 8x8 grid).

HIGH

Specifies the highest possible density for the grid at a given level. HIGH equates to 256 cells (a 16x16 grid).

Using level names allows you to specify the levels in any order and to omit levels. If you use the name for any level, you must use the name of any other level that you specify. If you omit a level, its density defaults to MEDIUM.

If an invalid density is specified, an error is raised.

CELLS_PER_OBJECT = n

Specifies the number of tessellation cells per object that can be used for a single spatial object in the index by the tessellation process. n can be any integer between 1 and 8192, inclusive. If an invalid number is passed or the number is larger than the maximum number of cells for the specified tessellation, an error is raised.

CELLS_PER_OBJECT has the following default values:

USING option	Default Cells per Object
GEOMETRY_GRID	**16**
GEOMETRY_AUTO_GRID	**8**
GEOGRAPHY_GRID	**16**
GEOGRAPHY_AUTO_GRID	**12**

At the top level, if an object covers more cells than specified by n, the indexing uses as many cells as necessary to provide a complete top-level tessellation. In such cases, an object might receive more than the specified number of cells. In this case, the maximum number is the number of cells generated by the top-level grid, which depends on the density.

The CELLS_PER_OBJECT value is used by the cells-per-object tessellation rule. For information about the tessellation rules, see Spatial Indexes Overview.

PAD_INDEX = { ON | OFF }

Specifies index padding. The default is OFF.

ON

Indicates that the percentage of free space that is specified by fillfactor is applied to the intermediate-level pages of the index.

OFF or fillfactor is not specified

Indicates that the intermediate-level pages are filled to near capacity, leaving sufficient space for at least one row of the maximum size the index can have, considering the set of keys on the intermediate pages.

The PAD_INDEX option is useful only when FILLFACTOR is specified, because PAD_INDEX uses the percentage specified by FILLFACTOR. If the percentage specified for FILLFACTOR is not large enough to allow for one row, the Database Engine internally overrides the percentage to allow for the minimum. The number of rows on an intermediate index page is never less than two, regardless of how low the value of fillfactor.

FILLFACTOR = fillfactor

Specifies a percentage that indicates how full the Database Engine should make the leaf level

of each index page during index creation or rebuild. fillfactor must be an integer value from 1 to 100. The default is 0. If fillfactor is 100 or 0, the Database Engine creates indexes with leaf pages filled to capacity.

Note

Fill factor values 0 and 100 are the same in all respects.

The FILLFACTOR setting applies only when the index is created or rebuilt. The Database Engine does not dynamically keep the specified percentage of empty space in the pages. To view the fill factor setting, use the sys.indexes catalog view.

Important

Creating a clustered index with a FILLFACTOR less than 100 affects the amount of storage space the data occupies because the Database Engine redistributes the data when it creates the clustered index.

For more information, see Fill Factor.

SORT_IN_TEMPDB = { ON | OFF }

Specifies whether to store temporary sort results in tempdb. The default is OFF.

ON

The intermediate sort results that are used to build the index are stored in tempdb. This may reduce the time required to create an index if tempdb is on a different set of disks than the user database. However, this increases the amount of disk space that is used during the index build.

OFF

The intermediate sort results are stored in the same database as the index.

In addition to the space required in the user database to create the index, tempdb must have about the same amount of additional space to hold the intermediate sort results. For more information, see tempdb and Index Creation.

IGNORE_DUP_KEY = OFF

Has no effect for spatial indexes because the index type is never unique. Do not set this option to ON, or else an error is raised.

STATISTICS_NORECOMPUTE = { ON | OFF }

Specifies whether distribution statistics are recomputed. The default is OFF.

ON

Out-of-date statistics are not automatically recomputed.

OFF

Automatic statistics updating are enabled.

To restore automatic statistics updating, set the STATISTICS_NORECOMPUTE to OFF, or execute UPDATE STATISTICS without the NORECOMPUTE clause.

Important

Disabling automatic recomputation of distribution statistics may prevent the query optimizer from picking optimal execution plans for queries involving the table.

DROP_EXISTING = { ON | OFF }

Specifies that the named, preexisting spatial index is dropped and rebuilt. The default is OFF.

ON

The existing index is dropped and rebuilt. The index name specified must be the same as a currently existing index; however, the index definition can be modified. For example, you can specify different columns, sort order, partition scheme, or index options.

OFF

An error is displayed if the specified index name already exists.

The index type cannot be changed by using DROP_EXISTING.

ONLINE = OFF

Specifies that underlying tables and associated indexes are not available for queries and data modification during the index operation. In this version of SQL Server, online index builds are not supported for spatial indexes. If this option is set to ON for a spatial index, an error is raised. Either omit the ONLINE option or set ONLINE to OFF.

An offline index operation that creates, rebuilds, or drops a spatial index, acquires a Schema modification (Sch-M) lock on the table. This prevents all user access to the underlying table for the duration of the operation.

Note

Online index operations are not available in every edition of Microsoft SQL Server. For a list of features that are supported by the editions of SQL Server, see Features Supported by the Editions of SQL Server 2012.

ALLOW_ROW_LOCKS = { ON | OFF }

Specifies whether row locks are allowed. The default is ON.

ON

Row locks are allowed when accessing the index. The Database Engine determines when row locks are used.

OFF

Row locks are not used.

ALLOW_PAGE_LOCKS = { ON | OFF }

Specifies whether page locks are allowed. The default is ON.

ON

Page locks are allowed when accessing the index. The Database Engine determines when page locks are used.

OFF

Page locks are not used.

MAXDOP = max_degree_of_parallelism

Overrides the `max degree of parallelism` configuration option for the duration of the index operation. Use MAXDOP to limit the number of processors used in a parallel plan execution. The maximum is 64 processors.

iImportant

Although the MAXDOP option is syntactically supported, CREATE SPATIAL INDEX currently always uses only a single processor.

max_degree_of_parallelism can be:

1

Suppresses parallel plan generation.

>1

Restricts the maximum number of processors used in a parallel index operation to the specified number or fewer based on the current system workload.

0 (default)

Uses the actual number of processors or fewer based on the current system workload.

For more information, see Configuring Parallel Index Operations.

Note

Parallel index operations are not available in every edition of Microsoft SQL Server. For a list of features that are supported by the editions of SQL Server, see Features Supported by the Editions of SQL Server 2012.

DATA_COMPRESSION = {NONE | ROW | PAGE}

Determines the level of data compression used by the index.

NONE

No compression used on data by the index

ROW

Row compression used on data by the index

PAGE

Page compression used on data by the index

Remarks

For an introduction to spatial indexing in SQL Server, see Spatial Indexes Overview.

Every option can be specified only once per CREATE SPATIAL INDEX statement. Specifying a duplicate of any option raises an error.

You can create up to 249 spatial indexes on each spatial column in a table. Creating more than one spatial index on specific spatial column can be useful, for example, to index different tessellation parameters in a single column.

Important
There are a number of other restrictions on creating a spatial index. For more information, see Spatial Indexes Overview.

An index build cannot make use of available process parallelism.

Methods Supported on Spatial Indexes

Under certain conditions, spatial indexes support a number of set-oriented geometry methods. For more information, see Spatial Indexes Overview.

Spatial Indexes and Partitioning

By default, if a spatial index is created on a partitioned table, the index is partitioned according to the partition scheme of the table. This assures that index data and the related row are stored in the same partition.

In this case, to alter the partition scheme of the base table, you would have to drop the spatial index before you can repartition the base table. To avoid this restriction, when you are creating a spatial index, you can specify the "ON filegroup" option. For more information, see "Spatial Indexes and Filegroups," later in this topic.

Spatial Indexes and Filegroups

By default, spatial indexes are partitioned to the same filegroups as the table on which the index is specified. This can be overridden by using the filegroup specification:

[ON { filegroup_name | "default" }]

If you specify a filegroup for a spatial index, the index is placed on that filegroup, regardless of the partitioning scheme of the table.

Catalog Views for Spatial Indexes

The following catalog views are specific to spatial indexes:

sys.spatial_indexes

Represents the main index information of the spatial indexes.

sys.spatial_index_tessellations

Represents the information about the tessellation scheme and parameters of each of the spatial indexes.

Additional Remarks About Creating Indexes

For more information about creating indexes, see the "Remarks" section in CREATE INDEX (Transact-SQL).

Permissions

The user must have ALTER permission on the table or view, or be a member of the sysadmin fixed server role or the db_ddladmin and db_owner fixed database roles.

Examples

A. Creating a spatial index on a geometry column

The following example creates a table named `SpatialTable` that contains a **geometry** type column, `geometry_col`. The example then creates a spatial index, `SIndx_SpatialTable_geometry_col1`, on the `geometry_col`. The example uses the default tessellation scheme and specifies the bounding box.

```
CREATE TABLE SpatialTable(id int primary key, geometry_col geometry);
CREATE SPATIAL INDEX SIndx_SpatialTable_geometry_col1
   ON SpatialTable(geometry_col)
   WITH ( BOUNDING_BOX = ( 0, 0, 500, 200 ) );
```

B. Creating a spatial index on a geometry column

The following example creates a second spatial index, `SIndx_SpatialTable_geometry_col2`, on the `geometry_col` in the `SpatialTable` table. The example specifies `GEOMETRY_GRID` as the tessellation scheme. The example also specifies the bounding box, different densities on different grid levels, and 64 cells per object. The example also sets the index padding to `ON`.

```
CREATE SPATIAL INDEX SIndx_SpatialTable_geometry_col2
   ON SpatialTable(geometry_col)
   USING GEOMETRY_GRID
   WITH (
    BOUNDING_BOX = ( xmin=0, ymin=0, xmax=500, ymax=200 ),
    GRIDS = (LOW, LOW, MEDIUM, HIGH),
    CELLS_PER_OBJECT = 64,
    PAD_INDEX  = ON );
```

C. Creating a spatial index on a geometry column

The following example creates a third spatial index, `SIndx_SpatialTable_geometry_col3`, on the `geometry_col` in the `SpatialTable` table. The example uses the default tessellation scheme. The example specifies the bounding box and uses different cell densities on the third and fourth levels, while using the default number of cells per object.

```
CREATE SPATIAL INDEX SIndx_SpatialTable_geometry_col3
   ON SpatialTable(geometry_col)
   WITH (
    BOUNDING_BOX = ( 0, 0, 500, 200 ),
    GRIDS = ( LEVEL_4 = HIGH, LEVEL_3 = MEDIUM ) );
```

D. Changing an option that is specific to spatial indexes

The following example rebuilds the spatial index created in the preceding example, `SIndx_SpatialTable_geography_col3`, by specifying a new `LEVEL_3` density with DROP_EXISTING = ON.

```
CREATE SPATIAL INDEX SIndx_SpatialTable_geography_col3
   ON SpatialTable(geography_col)
   WITH ( BOUNDING_BOX = ( 0, 0, 500, 200 ),
        GRIDS = ( LEVEL_3 = LOW ),
        DROP_EXISTING = ON );
```

E. Creating a spatial index on a geography column

The following example creates a table named `SpatialTable2` that contains a **geography** type column, `geography_col`. The example then creates a spatial index, `SIndx_SpatialTable_geography_col1`, on the `geography_col`. The example uses the default parameters values of the GEOGRAPHY_AUTO_GRID tessellation scheme.

```
CREATE TABLE SpatialTable2(id int primary key, object GEOGRAPHY);
CREATE SPATIAL INDEX SIndx_SpatialTable_geography_col1
   ON SpatialTable2(object);
```

Note

For geography grid indexes, a bounding box cannot be specified.

F. Creating a spatial index on a geography column

The following example creates a second spatial index, `SIndx_SpatialTable_geography_col2`, on the `geography_col` in the `SpatialTable2` table. The example specifies `GEOGRAPHY_GRID` as the tessellation scheme. The example also specifies different grid densities on different levels and 64 cells per object. The example also sets the index padding to `ON`.

```
CREATE SPATIAL INDEX SIndx_SpatialTable_geography_col2
   ON SpatialTable2(object)
   USING GEOGRAPHY_GRID
   WITH (
    GRIDS = (MEDIUM, LOW, MEDIUM, HIGH ),
    CELLS_PER_OBJECT = 64,
    PAD_INDEX  = ON );
```

G. Creating a spatial index on a geography column

The example then creates a third spatial index, `SIndx_SpatialTable_geography_col3`, on the `geography_col` in the `SpatialTable2` table. The example uses the default tessellation scheme, GEOGRAPHY_GRID, and the default CELLS_PER_OBJECT value (16).

```
CREATE SPATIAL INDEX SIndx_SpatialTable_geography_col3
   ON SpatialTable2(object)
   WITH ( GRIDS = ( LEVEL_3 = HIGH, LEVEL_2 = HIGH ) );
```

See Also

ALTER INDEX (Transact-SQL)
CREATE INDEX (Transact-SQL)
CREATE PARTITION FUNCTION
CREATE PARTITION SCHEME
CREATE STATISTICS
CREATE TABLE
Data Types
DBCC SHOW_STATISTICS
DROP INDEX
EVENTDATA (Transact-SQL)
sys.index_columns
sys.indexes
sys.spatial_index_tessellations (Transact-SQL)
sys.spatial_indexes (Transact-SQL)
Spatial Indexes Overview
Working with Spatial Indexes

CREATE STATISTICS

Creates query optimization statistics, including filtered statistics, on one or more columns of a table or indexed view. For most queries, the query optimizer already generates the necessary statistics for a high-quality query plan; in a few cases, you need to create additional statistics with CREATE STATISTICS or modify the query design to improve query performance.

Filtered statistics can improve query performance for queries that select from well-defined subsets of data. Filtered statistics use a filter predicate in the WHERE clause to select the subset of data that is included in the statistics. CREATE STATISTICS can use tempdb to sort the sample of rows for building statistics.

For more information about statistics, including when to use CREATE STATISTICS, see Using Statistics to Improve Query Performance.

 Transact-SQL Syntax Conventions

Syntax

```
CREATE STATISTICS statistics_name
ON { table_or_indexed_view_name } ( column [ ,...n ] )
   [ WHERE <filter_predicate> ]
   [ WITH
      [ [ FULLSCAN
        | SAMPLE number { PERCENT | ROWS }
        | STATS_STREAM = stats_stream ] [ , ] ]
      [ NORECOMPUTE ]
   ] ;

<filter_predicate> ::=
   <conjunct> [AND <conjunct>]

<conjunct> ::=
   <disjunct> | <comparison>

<disjunct> ::=
      column_name IN (constant ,...)

<comparison> ::=
      column_name <comparison_op> constant

<comparison_op> ::=
   IS | IS NOT | = | <> | != | > | >= | !> | < | <= | !<
```

Arguments

statistics_name

Is the name of the statistics to create.

table_or_indexed_view_name

Is the name of the table or indexed view to create the statistics on. Statistics can be created on tables or indexed views in another database by specifying a qualified table name.

column [,...n]

Specifies the key column or list of key columns to create the statistics on. You can specify any column that can be specified as an index key column with the following exceptions:

- **Xml**, full-text, and FILESTREAM columns cannot be specified.
- Computed columns can be specified only if the ARITHABORT and QUOTED_IDENTIFIER

database settings are ON.

- CLR user-defined type columns can be specified if the type supports binary ordering. Computed columns defined as method invocations of a user-defined type column can be specified if the methods are marked deterministic.

WHERE <filter_predicate>

Specifies an expression for selecting a subset of rows to include when creating the statistics object. Statistics that are created with a filter predicate are called filtered statistics. The filter predicate uses simple comparison logic and cannot reference a computed column, a UDT column, a spatial data type column, or a **hierarchyID** data type column. Comparisons using NULL literals are not allowed with the comparison operators. Use the IS NULL and IS NOT NULL operators instead.

Here are some examples of filter predicates for the Production.BillOfMaterials table:

```
WHERE StartDate > '20000101' AND EndDate <= '20000630'
WHERE ComponentID IN (533, 324, 753)
WHERE StartDate IN ('20000404', '20000905') AND EndDate IS
NOT NULL
```

For more information about filter predicates, see Filtered Index Design Guidelines.

FULLSCAN

Compute statistics by scanning all rows in the table or indexed view. FULLSCAN and SAMPLE 100 PERCENT have the same results. FULLSCAN cannot be used with the SAMPLE option.

SAMPLE number { PERCENT | ROWS }

Specifies the approximate percentage or number of rows in the table or indexed view for the query optimizer to use when it creates statistics. For PERCENT, number can be from 0 through 100 and for ROWS, number can be from 0 to the total number of rows. The actual percentage or number of rows the query optimizer samples might not match the percentage or number specified. For example, the query optimizer scans all rows on a data page.

SAMPLE is useful for special cases in which the query plan, based on default sampling, is not optimal. In most situations, it is not necessary to specify SAMPLE because the query optimizer already uses sampling and determines the statistically significant sample size by default, as required to create high-quality query plans.

SAMPLE cannot be used with the FULLSCAN option. When neither SAMPLE nor FULLSCAN is specified, the query optimizer uses sampled data and computes the sample size by default.

We recommend against specifying 0 PERCENT or 0 ROWS. When 0 PERCENT or ROWS is specified, the statistics object is created but does not contain statistics data.

NORECOMPUTE

Disable the automatic statistics update option, AUTO_STATISTICS_UPDATE, for *statistics_name*. If this option is specified, the query optimizer will complete any in-progress statistics updates for *statistics_name* and disable future updates.

To re-enable statistics updates, remove the statistics with DROP STATISTICS and then run CREATE STATISTICS without the NORECOMPUTE option.

Warning

Using this option can produce suboptimal query plans. We recommend using this option sparingly, and then only by a qualified system administrator.

For more information about the AUTO_STATISTICS_UPDATE option, see ALTER DATABASE SET Options (Transact-SQL). For more information about disabling and re-enabling statistics updates, see Using Statistics to Improve Query Performance.

STATS_STREAM = stats_stream

Identified for informational purposes only. Not supported. Future compatibility is not guaranteed.

Remarks

You can list up to 32 columns per statistics object.

When to Use CREATE STATISTICS

For more information about when to use CREATE STATISTICS, see Using Statistics to Improve Query Performance.

Referencing Dependencies for Filtered Statistics

The sys.sql_expression_dependencies catalog view tracks each column in the filtered statistics predicate as a referencing dependency. Consider the operations that you perform on table columns before creating filtered statistics because you cannot drop, rename, or alter the definition of a table column that is defined in a filtered statistics predicate.

Permissions

Requires ALTER TABLE permission, or the user must be the owner of the table or indexed view, or the user must be a member of the **db_ddladmin** fixed database role.

Examples

A. Using CREATE STATISTICS with SAMPLE number PERCENT

The following example creates the `ContactMail1` statistics, using a random sample of 5 percent of the `ContactID` and `EmailAddress` columns of the `Contact` table of the `AdventureWorks` database.

```
USE AdventureWorks2012;
GO
CREATE STATISTICS ContactMail1
    ON Person.Person (BusinessEntityID, EmailPromotion)
    WITH SAMPLE 5 PERCENT;
```

B. Using CREATE STATISTICS with FULLSCAN and NORECOMPUTE

The following example creates the `ContactMail2` statistics for all rows in the `ContactID` and `EmailAddress` columns of the `Contact` table and disables automatic recomputing of statistics.

```
CREATE STATISTICS NamePurchase
    ON AdventureWorks2012.Person.Person (BusinessEntityID, EmailPromotion)
    WITH FULLSCAN, NORECOMPUTE;
```

C. Using CREATE STATISTICS to create filtered statistics

The following example creates the filtered statistics `ContactPromotion1`. The Database Engine samples 50 percent of the data and then selects the rows with `EmailPromotion` equal to 2.

```
CREATE STATISTICS NamePurchase
    ON AdventureWorks2012.Person.Person (BusinessEntityID, EmailPromotion)
    WITH FULLSCAN, NORECOMPUTE;
```

See Also

Using Statistics to Improve Query Performance
UPDATE STATISTICS (Transact-SQL)
sp_updatestats (Transact-SQL)
DBCC SHOW_STATISTICS (Transact-SQL)
DROP STATISTICS (Transact-SQL)
sys.stats (Transact-SQL)
sys.stats_columns (Transact-SQL)

CREATE SYMMETRIC KEY

Generates a symmetric key and specifies its properties.

 Transact-SQL Syntax Conventions

Syntax

```
CREATE SYMMETRIC KEY key_name [ AUTHORIZATION owner_name ]
   [ FROM PROVIDER Provider_Name ]
   WITH <key_options> [ , ... n ]
      |
   ENCRYPTION BY <encrypting_mechanism> [ , ... n ]
<key_options> ::=
   KEY_SOURCE = 'pass_phrase'
   |
   ALGORITHM = <algorithm>
```

|

IDENTITY_VALUE = **'identity_phrase'**

|

PROVIDER_KEY_NAME = **'key_name_in_provider'**

|

CREATION_DISPOSITION = {CREATE_NEW | OPEN_EXISTING }

<algorithm> ::=

DES | TRIPLE_DES | TRIPLE_DES_3KEY | RC2 | RC4 | RC4_128

| DESX | AES_128 | AES_192 | AES_256

<encrypting_mechanism> ::=

CERTIFICATE **certificate_name**

|

PASSWORD = **'password'**

|

SYMMETRIC KEY **symmetric_key_name**

|

ASYMMETRIC KEY **asym_key_name**

Arguments

Key_name

Specifies the unique name by which the symmetric key is known in the database. The names of temporary keys should begin with one number (#) sign. For example, **#temporaryKey900007**. You cannot create a symmetric key that has a name that starts with more than one #. You cannot create a temporary symmetric key using an EKM provider.

AUTHORIZATION owner_name

Specifies the name of the database user or application role that will own this key.

FROM PROVIDER Provider_Name

Specifies an Extensible Key Management (EKM) provider and name. The key is not exported from the EKM device. The provider must be defined first using the CREATE PROVIDER statement. For more information about creating external key providers, see Understanding Extensible Key Management (EKM).

Note

This option is not available in a contained database.

KEY_SOURCE = 'pass_phrase'

Specifies a pass phrase from which to derive the key.

IDENTITY_VALUE = 'identity_phrase'

Specifies an identity phrase from which to generate a GUID for tagging data that is encrypted with a temporary key.

PROVIDER_KEY_NAME='key_name_in_provider'

Specifies the name referenced in the Extensible Key Management provider.

Note

This option is not available in a contained database.

CREATION_DISPOSITION = CREATE_NEW

Creates a new key can on the Extensible Key Management device. If a key already exists on the device, the statement fails with error.

CREATION_DISPOSITION = OPEN_EXISTING

Maps a SQL Server symmetric key to an existing Extensible Key Management key. If CREATION_DISPOSITION = OPEN_EXISTING is not provided, this defaults to CREATE_NEW.

certificate_name

Specifies the name of the certificate that will be used to encrypt the symmetric key. The certificate must already exist in the database.

'password'

Specifies a password from which to derive a TRIPLE_DES key with which to secure the symmetric key. password must meet the Windows password policy requirements of the computer that is running the instance of SQL Server. You should always use strong passwords.

symmetric_key_name

Specifies a symmetric key to be used to encrypt the key that is being created. The specified key must already exist in the database, and the key must be open.

asym_key_name

Specifies an asymmetric key to be used to encrypt the key that is being created. This asymmetric key must already exist in the database.

Remarks

When a symmetric key is created, the symmetric key must be encrypted by using at least one of the following: certificate, password, symmetric key, asymmetric key, or PROVIDER. The key can have more than one encryption of each type. In other words, a single symmetric key can be encrypted by using multiple certificates, passwords, symmetric keys, and asymmetric keys at the same time.

Caution

When a symmetric key is encrypted with a password instead of the public key of the database master key, the TRIPLE DES encryption algorithm is used. Because of this, keys

that are created with a strong encryption algorithm, such as AES, are themselves secured by a weaker algorithm.

The optional password can be used to encrypt the symmetric key before distributing the key to multiple users.

Temporary keys are owned by the user that creates them. Temporary keys are only valid for the current session.

IDENTITY_VALUE generates a GUID with which to tag data that is encrypted with the new symmetric key. This tagging can be used to match keys to encrypted data. The GUID generated by a specific phrase will always be the same. After a phrase has been used to generate a GUID, the phrase cannot be reused as long as there is at least one session that is actively using the phrase. IDENTITY_VALUE is an optional clause; however, we recommend using it when you are storing data encrypted with a temporary key.

There is no default encryption algorithm.

Important

We do not recommend using the RC4 and RC4_128 stream ciphers to protect sensitive data. SQL Server does not further encode the encryption performed with such keys.

Information about symmetric keys is visible in the sys.symmetric_keys catalog view.

Symmetric keys cannot be encrypted by symmetric keys created from the encryption provider.

Clarification regarding DES algorithms:

- DESX was incorrectly named. Symmetric keys created with ALGORITHM = DESX actually use the TRIPLE DES cipher with a 192-bit key. The DESX algorithm is not provided. This feature will be removed in a future version of Microsoft SQL Server. Avoid using this feature in new development work, and plan to modify applications that currently use this feature.
- Symmetric keys created with ALGORITHM = TRIPLE_DES_3KEY use TRIPLE DES with a 192-bit key.
- Symmetric keys created with ALGORITHM = TRIPLE_DES use TRIPLE DES with a 128-bit key.

Deprecation of the RC4 algorithm:

Repeated use of the same RC4 or RC4_128 KEY_GUID on different blocks of data will result in the same RC4 key because SQL Server does not provide a salt automatically. Using the same RC4 key repeatedly is a well known error that will result in very weak encryption. Therefore we have deprecated the RC4 and RC4_128 keywords. This feature will be removed in a future version of Microsoft SQL Server. Do not use this feature in new development work, and modify applications that currently use this feature as soon as possible.

Warning

The RC4 algorithm is only supported for backward compatibility. New material can only be encrypted using RC4 or RC4_128 when the database is in compatibility level 90 or 100. (Not recommended.) Use a newer algorithm such as one of the AES algorithms instead. In SQL Server 2012 material encrypted using RC4 or RC4_128 can be decrypted in any compatibility level.

Permissions

Requires ALTER ANY SYMMETRIC KEY permission on the database. If AUTHORIZATION is specified, requires IMPERSONATE permission on the database user or ALTER permission on the application role. If encryption is by certificate or asymmetric key, requires VIEW DEFINITION permission on the certificate or asymmetric key. Only Windows logins, SQL Server logins, and application roles can own symmetric keys. Groups and roles cannot own symmetric keys.

Examples

A. Creating a symmetric key

The following example creates a symmetric key called `JanainaKey09` by using the `AES 256` algorithm, and then encrypts the new key with certificate `Shipping04`.

```
CREATE SYMMETRIC KEY JanainaKey09 WITH ALGORITHM = AES_256
    ENCRYPTION BY CERTIFICATE Shipping04;
GO
```

B. Creating a temporary symmetric key

The following example creates a temporary symmetric key called `#MarketingXXV` from the pass phrase: `The square of the hypotenuse is equal to the sum of the squares of the sides`. The key is provisioned with a GUID that is generated from the string `Pythagoras` and encrypted with certificate `Marketing25`.

```
CREATE SYMMETRIC KEY #MarketingXXV
     WITH ALGORITHM = AES_128,
     KEY_SOURCE
     = 'The square of the hypotenuse is equal to the sum of the squares of
the sides',
     IDENTITY_VALUE = 'Pythagoras'
     ENCRYPTION BY CERTIFICATE Marketing25;
GO
```

C. Creating a symmetric key using an Extensible Key Management (EKM) device

The following example creates a symmetric key called `MySymKey` by using a provider called `MyEKMProvider` and a key name of `KeyForSensitiveData`. It assigns authorization to `User1` and assumes that the system administrator has already registered the provider called `MyEKMProvider` in SQL Server.

```
CREATE SYMMETRIC KEY MySymKey
AUTHORIZATION User1
FROM PROVIDER EKMProvider
WITH
```

```
PROVIDER_KEY_NAME='KeyForSensitiveData',
CREATION_DISPOSITION=OPEN_EXISTING;
GO
```

See Also

Choosing an Encryption Algorithm
ALTER SYMMETRIC KEY (Transact-SQL)
DROP SYMMETRIC KEY (Transact-SQL)
Encryption Hierarchy
sys.symmetric_keys (Transact-SQL)
Understanding Extensible Key Management (EKM)

CREATE SYNONYM

Creates a new synonym.

 Transact-SQL Syntax Conventions

Syntax

CREATE SYNONYM [**schema_name_1.**] **synonym_name** FOR <object>

<object> :: =
{
  [**server_name.**[**database_name**] **.** [**schema_name_2**].| **database_name** **.** [**schema_name_2**].| **schema_name_2.**] **object_name**
}

Arguments

schema_name_1

Specifies the schema in which the synonym is created. If schema is not specified, SQL Server uses the default schema of the current user.

synonym_name

Is the name of the new synonym.

server_name

Is the name of the server on which base object is located.

database_name

Is the name of the database in which the base object is located. If database_name is not specified, the name of the current database is used.

schema_name_2

Is the name of the schema of the base object. If schema_name is not specified the default schema of the current user is used.

object_name

Is the name of the base object that the synonym references.

Remarks

The base object need not exist at synonym create time. SQL Server checks for the existence of the base object at run time.

Synonyms can be created for the following types of objects:

Assembly (CLR) Stored Procedure	Assembly (CLR) Table-valued Function
Assembly (CLR) Scalar Function	Assembly Aggregate (CLR) Aggregate Functions
Replication-filter-procedure	Extended Stored Procedure
SQL Scalar Function	SQL Table-valued Function
SQL Inline-table-valued Function	SQL Stored Procedure
View	Table[1] (User-defined)

1 Includes local and global temporary tables

Four-part names for function base objects are not supported.

Synonyms can be created, dropped and referenced in dynamic SQL.

Permissions

To create a synonym in a given schema, a user must have CREATE SYNONYM permission and either own the schema or have ALTER SCHEMA permission.

The CREATE SYNONYM permission is a grantable permission.

Note

You do not need permission on the base object to successfully compile the CREATE SYNONYM statement, because all permission checking on the base object is deferred until run time.

Examples

A. Creating a synonym for a local object

The following example first creates a synonym for the base object, `Product` in the `AdventureWorks2012` database, and then queries the synonym.

```
USE tempdb;
GO
-- Create a synonym for the Product table in AdventureWorks2012.
CREATE SYNONYM MyProduct
FOR AdventureWorks2012.Production.Product;
GO

-- Query the Product table by using the synonym.
USE tempdb;
GO
SELECT ProductID, Name
FROM MyProduct
WHERE ProductID < 5;
GO
```

Here is the result set.

```
----------------------
ProductID   Name
----------- -------------------------
1           Adjustable Race
2           Bearing Ball
3           BB Ball Bearing
4           Headset Ball Bearings
(4 row(s) affected)
```

B. Creating a synonym to remote object

In the following example, the base object, `Contact`, resides on a remote server named `Server_Remote`.

```
EXEC sp_addlinkedserver Server_Remote;
GO
USE tempdb;
GO
CREATE SYNONYM MyEmployee FOR
Server_Remote.AdventureWorks2012.HumanResources.Employee;
GO
```

C. Creating a synonym for a user-defined function

The following example creates a function named `dbo.OrderDozen` that increases order amounts to an even dozen units. The example then creates the synonym `dbo.CorrectOrder` for the `dbo.OrderDozen` function.

```
-- Creating the dbo.OrderDozen function
CREATE FUNCTION dbo.OrderDozen (@OrderAmt int)
RETURNS int
WITH EXECUTE AS CALLER
AS
BEGIN
IF @OrderAmt % 12 <> 0
BEGIN
    SET @OrderAmt +=  12 - (@OrderAmt % 12)
END
RETURN(@OrderAmt);
END;
GO

-- Using the dbo.OrderDozen function
DECLARE @Amt int
SET @Amt = 15
SELECT @Amt AS OriginalOrder, dbo.OrderDozen(@Amt) AS ModifiedOrder

-- Create a synonym dbo.CorrectOrder for the dbo.OrderDozen function.
CREATE SYNONYM dbo.CorrectOrder
FOR dbo.OrderDozen;
GO

-- Using the dbo.CorrectOrder synonym.
DECLARE @Amt int
SET @Amt = 15
SELECT @Amt AS OriginalOrder, dbo.CorrectOrder(@Amt) AS ModifiedOrder
```

See Also

GRANT (Transact-SQL)

GRANT (Transact-SQL)

EVENTDATA (Transact-SQL)

CREATE TABLE

Creates a new table in SQL Server 2012.

 Transact-SQL Syntax Conventions

Syntax

```
CREATE TABLE
   [ database_name . [ schema_name ] . | schema_name . ] table_name
   [ AS FileTable ]
   ( { <column_definition> | <computed_column_definition>
      | <column_set_definition> | [ <table_constraint> ] [ ,...n ] } )
   [ ON { partition_scheme_name ( partition_column_name ) | filegroup
      | "default" } ]
   [ { TEXTIMAGE_ON { filegroup | "default" } ]
   [ FILESTREAM_ON { partition_scheme_name | filegroup
      | "default" } ]
   [ WITH ( <table_option> [ ,...n ] ) ]
[ ; ]

<column_definition> ::=
column_name <data_type>
   [ FILESTREAM ]
   [ COLLATE collation_name ]
   [ NULL | NOT NULL ]
   [
      [ CONSTRAINT constraint_name ] DEFAULT constant_expression ]
    | [ IDENTITY [ ( seed ,increment ) ] [ NOT FOR REPLICATION ]
   ]
   [ ROWGUIDCOL ] [ <column_constraint> [ ...n ] ]
   [ SPARSE ]

<data type> ::=
[ type_schema_name . ] type_name
   [ ( precision [ , scale ] ) | max |
```

```
    [ { CONTENT | DOCUMENT } ] xml_schema_collection ) ]

<column_constraint> ::=
[ CONSTRAINT constraint_name ]
{    { PRIMARY KEY | UNIQUE }
      [ CLUSTERED | NONCLUSTERED ]
      [
         WITH FILLFACTOR = fillfactor
       | WITH ( < index_option > [ , ...n ] )
      ]
      [ ON { partition_scheme_name ( partition_column_name )
         | filegroup | "default" } ]
  | [ FOREIGN KEY ]
      REFERENCES [ schema_name . ] referenced_table_name [ ( ref_column ) ]
      [ ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
      [ ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
      [ NOT FOR REPLICATION ]
  | CHECK [ NOT FOR REPLICATION ] ( logical_expression )
}

<computed_column_definition> ::=
column_name AS computed_column_expression
[ PERSISTED [ NOT NULL ] ]
[
   [ CONSTRAINT constraint_name ]
   { PRIMARY KEY | UNIQUE }
      [ CLUSTERED | NONCLUSTERED ]
      [
         WITH FILLFACTOR = fillfactor
       | WITH ( <index_option> [ , ...n ] )
      ]
   | [ FOREIGN KEY ]
      REFERENCES referenced_table_name [ ( ref_column ) ]
      [ ON DELETE { NO ACTION | CASCADE } ]
      [ ON UPDATE { NO ACTION } ]
```

```
      [ NOT FOR REPLICATION ]
   | CHECK [ NOT FOR REPLICATION ] ( logical_expression )
   [ ON { partition_scheme_name ( partition_column_name )
      | filegroup | "default" } ]
]

<column_set_definition> ::=
column_set_name XML COLUMN_SET FOR ALL_SPARSE_COLUMNS

< table_constraint > ::=
[ CONSTRAINT constraint_name ]
{
   { PRIMARY KEY | UNIQUE }
      [ CLUSTERED | NONCLUSTERED ]
      (column [ ASC | DESC ] [ ,...n ] )
      [
         WITH FILLFACTOR = fillfactor
        |WITH ( <index_option> [ , ...n ] )
      ]
      [ ON { partition_scheme_name (partition_column_name)
         | filegroup | "default" } ]
   | FOREIGN KEY
      ( column [ ,...n ] )
      REFERENCES referenced_table_name [ ( ref_column [ ,...n ] ) ]
      [ ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
      [ ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
      [ NOT FOR REPLICATION ]
   | CHECK [ NOT FOR REPLICATION ] ( logical_expression )
}
<table_option> ::=
{
   [DATA_COMPRESSION = { NONE | ROW | PAGE }
    [ ON PARTITIONS ( { <partition_number_expression> | <range> }
    [ , ...n ] ) ]]
   [ FILETABLE_DIRECTORY = <directory_name> ]
```

[FILETABLE_COLLATE_FILENAME = { <collation_name> | database_default }]
[FILETABLE_PRIMARY_KEY_CONSTRAINT_NAME = <constraint_name>]
[FILETABLE_STREAMID_UNIQUE_CONSTRAINT_NAME = <constraint_name>]
[FILETABLE_FULLPATH_UNIQUE_CONSTRAINT_NAME = <constraint_name>]
}

<index_option> ::=

{
PAD_INDEX = { ON | OFF }
| FILLFACTOR = **fillfactor**
| IGNORE_DUP_KEY = { ON | OFF }
| STATISTICS_NORECOMPUTE = { ON | OFF }
| ALLOW_ROW_LOCKS = { ON | OFF}
| ALLOW_PAGE_LOCKS ={ ON | OFF}
| DATA_COMPRESSION = { NONE | ROW | PAGE }
[ON PARTITIONS **(** { <partition_number_expression> | <range> }
[**,** ...**n**] **)**]
}

<range> ::=

<partition_number_expression> TO <partition_number_expression>

Arguments

database_name

Is the name of the database in which the table is created. database_name must specify the name of an existing database. If not specified, database_name defaults to the current database. The login for the current connection must be associated with an existing user ID in the database specified by database_name, and that user ID must have CREATE TABLE permissions.

schema_name

Is the name of the schema to which the new table belongs.

table_name

Is the name of the new table. Table names must follow the rules for identifiers. table_name can be a maximum of 128 characters, except for local temporary table names (names prefixed with a single number sign (#)) that cannot exceed 116 characters.

AS FileTable

Creates the new table as a FileTable. You do not specify columns because a FileTable has a fixed schema. For more information about FileTables, see Storing Files with FileTables.

column_name

Is the name of a column in the table. Column names must follow the rules for identifiers and must be unique in the table. column_name can be up to 128 characters. column_name can be omitted for columns that are created with a **timestamp** data type. If column_name is not specified, the name of a **timestamp** column defaults to **timestamp**.

computed_column_expression

Is an expression that defines the value of a computed column. A computed column is a virtual column that is not physically stored in the table, unless the column is marked PERSISTED. The column is computed from an expression that uses other columns in the same table. For example, a computed column can have the definition: **cost** AS **price** * **qty**. The expression can be a noncomputed column name, constant, function, variable, and any combination of these connected by one or more operators. The expression cannot be a subquery or contain alias data types.

Computed columns can be used in select lists, WHERE clauses, ORDER BY clauses, or any other locations in which regular expressions can be used, with the following exceptions:

- A computed column cannot be used as a DEFAULT or FOREIGN KEY constraint definition or with a NOT NULL constraint definition. However, a computed column can be used as a key column in an index or as part of any PRIMARY KEY or UNIQUE constraint, if the computed column value is defined by a deterministic expression and the data type of the result is allowed in index columns.

  For example, if the table has integer columns **a** and **b**, the computed column **a+b** may be indexed, but computed column **a+DATEPART(dd, GETDATE())** cannot be indexed because the value may change in subsequent invocations.
- A computed column cannot be the target of an INSERT or UPDATE statement.

Note

Each row in a table can have different values for columns that are involved in a computed column; therefore, the computed column may not have the same value for each row.

Based on the expressions that are used, the nullability of computed columns is determined automatically by the Database Engine. The result of most expressions is considered nullable even if only nonnullable columns are present, because possible underflows or overflows also produce NULL results. Use the COLUMNPROPERTY function with the **AllowsNull** property to investigate the nullability of any computed column in a table. An expression that is nullable can be turned into a nonnullable one by specifying ISNULL with the check_expression constant, where the constant is a nonnull value substituted for any NULL result. REFERENCES permission on the type is required for computed columns based on common language runtime (CLR) user-defined type expressions.

PERSISTED

Specifies that the SQL Server Database Engine will physically store the computed values in the table, and update the values when any other columns on which the computed column depends are updated. Marking a computed column as PERSISTED lets you create an index on

a computed column that is deterministic, but not precise. For more information, see sp_spaceused (Transact-SQL). Any computed columns that are used as partitioning columns of a partitioned table must be explicitly marked PERSISTED. computed_column_expression must be deterministic when PERSISTED is specified.

ON { <partition_scheme> | filegroup | "default" }

Specifies the partition scheme or filegroup on which the table is stored. If <partition_scheme> is specified, the table is to be a partitioned table whose partitions are stored on a set of one or more filegroups specified in <partition_scheme>. If filegroup is specified, the table is stored in the named filegroup. The filegroup must exist within the database. If **"**default**"** is specified, or if ON is not specified at all, the table is stored on the default filegroup. The storage mechanism of a table as specified in CREATE TABLE cannot be subsequently altered.

ON {<partition_scheme> | filegroup | **"**default**"**} can also be specified in a PRIMARY KEY or UNIQUE constraint. These constraints create indexes. If filegroup is specified, the index is stored in the named filegroup. If **"**default**"** is specified, or if ON is not specified at all, the index is stored in the same filegroup as the table. If the PRIMARY KEY or UNIQUE constraint creates a clustered index, the data pages for the table are stored in the same filegroup as the index. If CLUSTERED is specified or the constraint otherwise creates a clustered index, and a <partition_scheme> is specified that differs from the <partition_scheme> or filegroup of the table definition, or vice-versa, only the constraint definition will be honored, and the other will be ignored.

Note

In this context, default is not a keyword. It is an identifier for the default filegroup and must be delimited, as in ON **"**default**"** or ON **[**default**]**. If **"**default**"** is specified, the QUOTED_IDENTIFIER option must be ON for the current session. This is the default setting. For more information, see SET QUOTED_IDENTIFIER (Transact-SQL).

Note

After you create a partitioned table, consider setting the LOCK_ESCALATION option for the table to AUTO. This can improve concurrency by enabling locks to escalate to partition (HoBT) level instead of the table. For more information, see ALTER TABLE (Transact-SQL).

TEXTIMAGE_ON { filegroup | "default" }

Indicates that the **text**, **ntext**, **image**, **xml**, **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, and CLR user-defined type columns (including geometry and geography) are stored on the specified filegroup.

TEXTIMAGE_ON is not allowed if there are no large value columns in the table. TEXTIMAGE_ON cannot be specified if <partition_scheme> is specified. If **"**default**"** is specified, or if TEXTIMAGE_ON is not specified at all, the large value columns are stored in the default filegroup. The storage of any large value column data specified in CREATE TABLE cannot be subsequently altered.

Note

In this context, default is not a keyword. It is an identifier for the default filegroup and must be delimited, as in TEXTIMAGE_ON "default" or TEXTIMAGE_ON [default]. If "default" is specified, the QUOTED_IDENTIFIER option must be ON for the current session. This is the default setting. For more information, see SET QUOTED_IDENTIFIER (Transact-SQL).

FILESTREAM_ON { partition_scheme_name | filegroup | "default" }

Specifies the filegroup for FILESTREAM data.

If the table contains FILESTREAM data and the table is partitioned, the FILESTREAM_ON clause must be included and must specify a partition scheme of FILESTREAM filegroups. This partition scheme must use the same partition function and partition columns as the partition scheme for the table; otherwise, an error is raised.

If the table is not partitioned, the FILESTREAM column cannot be partitioned. FILESTREAM data for the table must be stored in a single filegroup. This filegroup is specified in the FILESTREAM_ON clause.

If the table is not partitioned and the FILESTREAM_ON clause is not specified, the FILESTREAM filegroup that has the DEFAULT property set is used. If there is no FILESTREAM filegroup, an error is raised.

- As with ON and TEXTIMAGE_ON, the value set by using CREATE TABLE for FILESTREAM_ON cannot be changed, except in the following cases:
- A CREATE INDEX statement converts a heap into a clustered index. In this case, a different FILESTREAM filegroup, partition scheme, or NULL can be specified.
- A DROP INDEX statement converts a clustered index into a heap. In this case, a different FILESTREAM filegroup, partition scheme, or "default" can be specified.

The filegroup in the FILESTREAM_ON <filegroup> clause, or each FILESTREAM filegroup that is named in the partition scheme, must have one file defined for the filegroup. This file must be defined by using a CREATE DATABASE or ALTER DATABASE statement; otherwise, an error is raised.

For related FILESTREAM topics, see Designing and Implementing FILESTREAM Storage.

[type_schema_name.] type_name

Specifies the data type of the column, and the schema to which it belongs. The data type can be one of the following:

- A system data type.
- An alias type based on a SQL Server system data type. Alias data types are created with the CREATE TYPE statement before they can be used in a table definition. The NULL or NOT NULL assignment for an alias data type can be overridden during the CREATE TABLE statement. However, the length specification cannot be changed; the length for an alias data type cannot be specified in a CREATE TABLE statement.
- A CLR user-defined type. CLR user-defined types are created with the CREATE TYPE

statement before they can be used in a table definition. To create a column on CLR user-defined type, REFERENCES permission is required on the type.

If type_schema_name is not specified, the SQL Server Database Engine references type_name in the following order:

- The SQL Server system data type.
- The default schema of the current user in the current database.
- The **dbo** schema in the current database.

precision

Is the precision for the specified data type. For more information about valid precision values, see Precision, Scale, and Length.

scale

Is the scale for the specified data type. For more information about valid scale values, see Precision, Scale, and Length.

max

Applies only to the **varchar**, **nvarchar**, and **varbinary** data types for storing 2^31 bytes of character and binary data, and 2^30 bytes of Unicode data.

CONTENT

Specifies that each instance of the **xml** data type in column_name can contain multiple top-level elements. CONTENT applies only to the **xml** data type and can be specified only if xml_schema_collection is also specified. If not specified, CONTENT is the default behavior.

DOCUMENT

Specifies that each instance of the **xml** data type in column_name can contain only one top-level element. DOCUMENT applies only to the **xml** data type and can be specified only if xml_schema_collection is also specified.

xml_schema_collection

Applies only to the **xml** data type for associating an XML schema collection with the type. Before typing an **xml** column to a schema, the schema must first be created in the database by using CREATE XML SCHEMA COLLECTION.

DEFAULT

Specifies the value provided for the column when a value is not explicitly supplied during an insert. DEFAULT definitions can be applied to any columns except those defined as **timestamp**, or those with the IDENTITY property. If a default value is specified for a user-defined type column, the type should support an implicit conversion from constant_expression to the user-defined type. DEFAULT definitions are removed when the table is dropped. Only a constant value, such as a character string; a scalar function (either a system, user-defined, or CLR function); or NULL can be used as a default. To maintain compatibility with earlier versions of SQL Server, a constraint name can be assigned to a

DEFAULT.

constant_expression

Is a constant, NULL, or a system function that is used as the default value for the column.

IDENTITY

Indicates that the new column is an identity column. When a new row is added to the table, the Database Engine provides a unique, incremental value for the column. Identity columns are typically used with PRIMARY KEY constraints to serve as the unique row identifier for the table. The IDENTITY property can be assigned to **tinyint**, **smallint**, **int**, **bigint**, **decimal(p,0)**, or **numeric(p,0)** columns. Only one identity column can be created per table. Bound defaults and DEFAULT constraints cannot be used with an identity column. Both the seed and increment or neither must be specified. If neither is specified, the default is (1,1).

seed

Is the value used for the very first row loaded into the table.

increment

Is the incremental value added to the identity value of the previous row loaded.

NOT FOR REPLICATION

In the CREATE TABLE statement, the NOT FOR REPLICATION clause can be specified for the IDENTITY property, FOREIGN KEY constraints, and CHECK constraints. If this clause is specified for the IDENTITY property, values are not incremented in identity columns when replication agents perform inserts. If this clause is specified for a constraint, the constraint is not enforced when replication agents perform insert, update, or delete operations.

ROWGUIDCOL

Indicates that the new column is a row GUID column. Only one **uniqueidentifier** column per table can be designated as the ROWGUIDCOL column. Applying the ROWGUIDCOL property enables the column to be referenced using $ROWGUID. The ROWGUIDCOL property can be assigned only to a **uniqueidentifier** column. User-defined data type columns cannot be designated with ROWGUIDCOL.

The ROWGUIDCOL property does not enforce uniqueness of the values stored in the column. ROWGUIDCOL also does not automatically generate values for new rows inserted into the table. To generate unique values for each column, either use the NEWID or NEWSEQUENTIALID function on INSERT statements or use these functions as the default for the column.

SPARSE

Indicates that the column is a sparse column. The storage of sparse columns is optimized for null values. Sparse columns cannot be designated as NOT NULL. For additional restrictions and more information about sparse columns, see Using Sparse Columns.

FILESTREAM

Valid only for **varbinary(max)** columns. Specifies FILESTREAM storage for the **varbinary(max)** BLOB data.

The table must also have a column of the **uniqueidentifier** data type that has the ROWGUIDCOL attribute. This column must not allow null values and must have either a UNIQUE or PRIMARY KEY single-column constraint. The GUID value for the column must be supplied either by an application when inserting data, or by a DEFAULT constraint that uses the NEWID () function.

The ROWGUIDCOL column cannot be dropped and the related constraints cannot be changed while there is a FILESTREAM column defined for the table. The ROWGUIDCOL column can be dropped only after the last FILESTREAM column is dropped.

When the FILESTREAM storage attribute is specified for a column, all values for that column are stored in a FILESTREAM data container on the file system.

COLLATE collation_name

Specifies the collation for the column. Collation name can be either a Windows collation name or an SQL collation name. collation_name is applicable only for columns of the **char**, **varchar**, **text**, **nchar**, **nvarchar**, and **ntext** data types. If not specified, the column is assigned either the collation of the user-defined data type, if the column is of a user-defined data type, or the default collation of the database.

For more information about the Windows and SQL collation names, see Windows Collation Name and SQL Collation Name.

For more information about the COLLATE clause, see COLLATE.

CONSTRAINT

Is an optional keyword that indicates the start of the definition of a PRIMARY KEY, NOT NULL, UNIQUE, FOREIGN KEY, or CHECK constraint.

constraint_name

Is the name of a constraint. Constraint names must be unique within the schema to which the table belongs.

NULL | NOT NULL

Determine whether null values are allowed in the column. NULL is not strictly a constraint but can be specified just like NOT NULL. NOT NULL can be specified for computed columns only if PERSISTED is also specified.

PRIMARY KEY

Is a constraint that enforces entity integrity for a specified column or columns through a unique index. Only one PRIMARY KEY constraint can be created per table.

UNIQUE

Is a constraint that provides entity integrity for a specified column or columns through a unique index. A table can have multiple UNIQUE constraints.

CLUSTERED | NONCLUSTERED

Indicate that a clustered or a nonclustered index is created for the PRIMARY KEY or UNIQUE constraint. PRIMARY KEY constraints default to CLUSTERED, and UNIQUE constraints default to NONCLUSTERED.

In a CREATE TABLE statement, CLUSTERED can be specified for only one constraint. If CLUSTERED is specified for a UNIQUE constraint and a PRIMARY KEY constraint is also specified, the PRIMARY KEY defaults to NONCLUSTERED.

FOREIGN KEY REFERENCES

Is a constraint that provides referential integrity for the data in the column or columns. FOREIGN KEY constraints require that each value in the column exists in the corresponding referenced column or columns in the referenced table. FOREIGN KEY constraints can reference only columns that are PRIMARY KEY or UNIQUE constraints in the referenced table or columns referenced in a UNIQUE INDEX on the referenced table. Foreign keys on computed columns must also be marked PERSISTED.

[schema_name .] referenced_table_name]

Is the name of the table referenced by the FOREIGN KEY constraint, and the schema to which it belongs.

(ref_column [,... n])

Is a column, or list of columns, from the table referenced by the FOREIGN KEY constraint.

ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT }

Specifies what action happens to rows in the table created, if those rows have a referential relationship and the referenced row is deleted from the parent table. The default is NO ACTION.

NO ACTION

The Database Engine raises an error and the delete action on the row in the parent table is rolled back.

CASCADE

Corresponding rows are deleted from the referencing table if that row is deleted from the parent table.

SET NULL

All the values that make up the foreign key are set to NULL if the corresponding row in the parent table is deleted. For this constraint to execute, the foreign key columns must be nullable.

SET DEFAULT

All the values that make up the foreign key are set to their default values if the corresponding row in the parent table is deleted. For this constraint to execute, all foreign key columns must have default definitions. If a column is nullable, and there is no explicit

default value set, NULL becomes the implicit default value of the column.

Do not specify CASCADE if the table will be included in a merge publication that uses logical records. For more information about logical records, see Grouping Changes to Related Rows with Logical Records.

ON DELETE CASCADE cannot be defined if an INSTEAD OF trigger ON DELETE already exists on the table.

For example, in the database, the **ProductVendor** table has a referential relationship with the **Vendor** table. The **ProductVendor.BusinessEntityID** foreign key references the **Vendor.BusinessEntityID** primary key.

If a DELETE statement is executed on a row in the **Vendor** table, and an ON DELETE CASCADE action is specified for **ProductVendor.BusinessEntityID**, the Database Engine checks for one or more dependent rows in the **ProductVendor** table. If any exist, the dependent rows in the **ProductVendor** table are deleted, and also the row referenced in the **Vendor** table.

Conversely, if NO ACTION is specified, the Database Engine raises an error and rolls back the delete action on the **Vendor** row if there is at least one row in the **ProductVendor** table that references it.

ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT }

Specifies what action happens to rows in the table altered when those rows have a referential relationship and the referenced row is updated in the parent table. The default is NO ACTION.

NO ACTION

The Database Engine raises an error, and the update action on the row in the parent table is rolled back.

CASCADE

Corresponding rows are updated in the referencing table when that row is updated in the parent table.

SET NULL

All the values that make up the foreign key are set to NULL when the corresponding row in the parent table is updated. For this constraint to execute, the foreign key columns must be nullable.

SET DEFAULT

All the values that make up the foreign key are set to their default values when the corresponding row in the parent table is updated. For this constraint to execute, all foreign key columns must have default definitions. If a column is nullable, and there is no explicit default value set, NULL becomes the implicit default value of the column.

Do not specify CASCADE if the table will be included in a merge publication that uses logical records. For more information about logical records, see Grouping Changes to Related Rows with Logical Records.

ON UPDATE CASCADE, SET NULL, or SET DEFAULT cannot be defined if an INSTEAD OF trigger ON UPDATE already exists on the table that is being altered.

For example, in the database, the **ProductVendor** table has a referential relationship with the **Vendor** table: **ProductVendor.BusinessEntity** foreign key references the **Vendor.BusinessEntityID** primary key.

If an UPDATE statement is executed on a row in the **Vendor** table, and an ON UPDATE CASCADE action is specified for **ProductVendor.BusinessEntityID**, the Database Engine checks for one or more dependent rows in the **ProductVendor** table. If any exist, the dependent rows in the **ProductVendor** table are updated, and also the row referenced in the **Vendor** table.

Conversely, if NO ACTION is specified, the Database Engine raises an error and rolls back the update action on the **Vendor** row if there is at least one row in the **ProductVendor** table that references it.

CHECK

Is a constraint that enforces domain integrity by limiting the possible values that can be entered into a column or columns. CHECK constraints on computed columns must also be marked PERSISTED.

logical_expression

Is a logical expression that returns TRUE or FALSE. Alias data types cannot be part of the expression.

column

Is a column or list of columns, in parentheses, used in table constraints to indicate the columns used in the constraint definition.

[ASC | DESC]

Specifies the order in which the column or columns participating in table constraints are sorted. The default is ASC.

partition_scheme_name

Is the name of the partition scheme that defines the filegroups onto which the partitions of a partitioned table will be mapped. The partition scheme must exist within the database.

[partition_column_name.]

Specifies the column against which a partitioned table will be partitioned. The column must match that specified in the partition function that partition_scheme_name is using in terms of data type, length, and precision. A computed columns that participates in a partition function must be explicitly marked PERSISTED.

Important

We recommend that you specify NOT NULL on the partitioning column of partitioned tables, and also nonpartitioned tables that are sources or targets of ALTER TABLE...SWITCH operations. Doing this

makes sure that any CHECK constraints on partitioning columns do not have to check for null values.

WITH FILLFACTOR = fillfactor

Specifies how full the Database Engine should make each index page that is used to store the index data. User-specified fillfactor values can be from 1 through 100. If a value is not specified, the default is 0. Fill factor values 0 and 100 are the same in all respects.

Important

Documenting WITH FILLFACTOR = fillfactor as the only index option that applies to PRIMARY KEY or UNIQUE constraints is maintained for backward compatibility, but will not be documented in this manner in future releases.

column_set_name XML COLUMN_SET FOR ALL_SPARSE_COLUMNS

Is the name of the column set. A column set is an untyped XML representation that combines all of the sparse columns of a table into a structured output. For more information about column sets, see Using Sparse Column Sets.

< table_option> ::=

Specifies one or more table options.

DATA_COMPRESSION

Specifies the data compression option for the specified table, partition number, or range of partitions. The options are as follows:

NONE

Table or specified partitions are not compressed.

ROW

Table or specified partitions are compressed by using row compression.

PAGE

Table or specified partitions are compressed by using page compression.

For more information about compression, see Creating Compressed Tables and Indexes.

ON PARTITIONS ({ <partition_number_expression> | <range> } [,...n])

Specifies the partitions to which the DATA_COMPRESSION setting applies. If the table is not partitioned, the ON PARTITIONS argument will generate an error. If the ON PARTITIONS clause is not provided, the DATA_COMPRESSION option will apply to all partitions of a partitioned table.

<partition_number_expression> can be specified in the following ways:

- Provide the partition number of a partition, for example: ON PARTITIONS (2).
- Provide the partition numbers for several individual partitions separated by commas, for example: ON PARTITIONS (1, 5).
- Provide both ranges and individual partitions, for example: ON PARTITIONS (2, 4, 6 TO 8)

<range> can be specified as partition numbers separated by the word TO, for example: ON PARTITIONS (6 TO 8).

To set different types of data compression for different partitions, specify the DATA_COMPRESSION option more than once, for example:

```
WITH
(
DATA_COMPRESSION = NONE ON PARTITIONS (1),
DATA_COMPRESSION = ROW ON PARTITIONS (2, 4, 6 TO 8),
DATA_COMPRESSION = PAGE ON PARTITIONS (3, 5)
)
```

<index_option> ::=

Specifies one or more index options. For a complete description of these options, see CREATE INDEX (Transact-SQL).

PAD_INDEX = { ON | OFF }

When ON, the percentage of free space specified by FILLFACTOR is applied to the intermediate level pages of the index. When OFF or a FILLFACTOR value it not specified, the intermediate level pages are filled to near capacity leaving enough space for at least one row of the maximum size the index can have, considering the set of keys on the intermediate pages. The default is OFF.

FILLFACTOR = fillfactor

Specifies a percentage that indicates how full the Database Engine should make the leaf level of each index page during index creation or alteration. fillfactor must be an integer value from 1 to 100. The default is 0. Fill factor values 0 and 100 are the same in all respects.

IGNORE_DUP_KEY = { ON | OFF }

Specifies the error response when an insert operation attempts to insert duplicate key values into a unique index. The IGNORE_DUP_KEY option applies only to insert operations after the index is created or rebuilt. The option has no effect when executing CREATE INDEX, ALTER INDEX, or UPDATE. The default is OFF.

ON

A warning message will occur when duplicate key values are inserted into a unique index. Only the rows violating the uniqueness constraint will fail.

OFF

An error message will occur when duplicate key values are inserted into a unique index. The entire INSERT operation will be rolled back.

IGNORE_DUP_KEY cannot be set to ON for indexes created on a view, non-unique indexes, XML indexes, spatial indexes, and filtered indexes.

To view IGNORE_DUP_KEY, use sys.indexes.

In backward compatible syntax, WITH IGNORE_DUP_KEY is equivalent to WITH IGNORE_DUP_KEY = ON.

STATISTICS_NORECOMPUTE = { ON | OFF }

When ON, out-of-date index statistics are not automatically recomputed. When OFF, automatic statistics updating are enabled. The default is OFF.

ALLOW_ROW_LOCKS = { ON | OFF }

When ON, row locks are allowed when you access the index. The Database Engine determines when row locks are used. When OFF, row locks are not used. The default is ON.

ALLOW_PAGE_LOCKS = { ON | OFF }

When ON, page locks are allowed when you access the index. The Database Engine determines when page locks are used. When OFF, page locks are not used. The default is ON.

FILETABLE_DIRECTORY = directory_name

Specifies the windows-compatible FileTable directory name. This name should be unique among all the FileTable directory names in the database. Uniqueness comparison is case-insensitive, regardless of collation settings. If this value is not specified, the name of the filetable is used.

FILETABLE_COLLATE_FILENAME = { collation_name | database_default }

Specifies the name of the collation to be applied to the **Name** column in the FileTable. The collation must be case-insensitive to comply with Windows file naming semantics. If this value is not specified, the database default collation is used. If the database default collation is case-sensitive, an error is raised and the CREATE TABLE operation fails.

collation_name

The name of a case-insensitive collation.

database_default

Specifies that the default collation for the database should be used. This collation must be case-insensitive.

FILETABLE_PRIMARY_KEY_CONSTRAINT_NAME = constraint_name

Specifies the name to be used for the primary key constraint that is automatically created on the FileTable. If this value is not specified, the system generates a name for the constraint.

FILETABLE_STREAMID_UNIQUE_CONSTRAINT_NAME = constraint_name

Specifies the name to be used for the unique constraint that is automatically created on the **stream_id** column in the FileTable. If this value is not specified, the system generates a name for the constraint.

FILETABLE_FULLPATH_UNIQUE_CONSTRAINT_NAME = constraint_name

Specifies the name to be used for the unique constraint that is automatically created on the

parent_path_locator and **name** columns in the FileTable. If this value is not specified, the system generates a name for the constraint.

Remarks

For information about the number of allowed tables, columns, constraints and indexes, see Maximum Capacity Specifications for SQL Server.

Space is generally allocated to tables and indexes in increments of one extent at a time. When the table or index is created, it is allocated pages from mixed extents until it has enough pages to fill a uniform extent. After it has enough pages to fill a uniform extent, another extent is allocated every time the currently allocated extents become full. For a report about the amount of space allocated and used by a table, execute **sp_spaceused**.

The Database Engine does not enforce an order in which DEFAULT, IDENTITY, ROWGUIDCOL, or column constraints are specified in a column definition.

When a table is created, the QUOTED IDENTIFIER option is always stored as ON in the metadata for the table, even if the option is set to OFF when the table is created.

Temporary Tables

You can create local and global temporary tables. Local temporary tables are visible only in the current session, and global temporary tables are visible to all sessions. Temporary tables cannot be partitioned.

Prefix local temporary table names with single number sign (#table_name), and prefix global temporary table names with a double number sign (##table_name).

SQL statements reference the temporary table by using the value specified for table_name in the CREATE TABLE statement, for example:

```
CREATE TABLE #MyTempTable (cola INT PRIMARY KEY);

INSERT INTO #MyTempTable VALUES (1);
```

If more than one temporary table is created inside a single stored procedure or batch, they must have different names.

If a local temporary table is created in a stored procedure or application that can be executed at the same time by several users, the Database Engine must be able to distinguish the tables created by the different users. The Database Engine does this by internally appending a numeric suffix to each local temporary table name. The full name of a temporary table as stored in the **sysobjects** table in **tempdb** is made up of the table name specified in the CREATE TABLE statement and the system-generated numeric suffix. To allow for the suffix, table_name specified for a local temporary name cannot exceed 116 characters.

Temporary tables are automatically dropped when they go out of scope, unless explicitly dropped by using DROP TABLE:

- A local temporary table created in a stored procedure is dropped automatically when the stored procedure is finished. The table can be referenced by any nested stored procedures executed by the stored procedure that created the table. The table cannot be referenced by the process that called the stored procedure that created the table.

- All other local temporary tables are dropped automatically at the end of the current session.
- Global temporary tables are automatically dropped when the session that created the table ends and all other tasks have stopped referencing them. The association between a task and a table is maintained only for the life of a single Transact-SQL statement. This means that a global temporary table is dropped at the completion of the last Transact-SQL statement that was actively referencing the table when the creating session ended.

A local temporary table created within a stored procedure or trigger can have the same name as a temporary table that was created before the stored procedure or trigger is called. However, if a query references a temporary table and two temporary tables with the same name exist at that time, it is not defined which table the query is resolved against. Nested stored procedures can also create temporary tables with the same name as a temporary table that was created by the stored procedure that called it. However, for modifications to resolve to the table that was created in the nested procedure, the table must have the same structure, with the same column names, as the table created in the calling procedure. This is shown in the following example.

```
CREATE PROCEDURE dbo.Test2
AS
    CREATE TABLE #t(x INT PRIMARY KEY);
    INSERT INTO #t VALUES (2);
    SELECT Test2Col = x FROM #t;
GO

CREATE PROCEDURE dbo.Test1
AS
    CREATE TABLE #t(x INT PRIMARY KEY);
    INSERT INTO #t VALUES (1);
    SELECT Test1Col = x FROM #t;
EXEC Test2;
GO

CREATE TABLE #t(x INT PRIMARY KEY);
INSERT INTO #t VALUES (99);
GO

EXEC
```

Here is the result set.

```
(1 row(s) affected)
```

```
Test1Col
-----------
1

(1 row(s) affected)

Test2Col
-----------
2
```

When you create local or global temporary tables, the CREATE TABLE syntax supports constraint definitions except for FOREIGN KEY constraints. If a FOREIGN KEY constraint is specified in a temporary table, the statement returns a warning message that states the constraint was skipped. The table is still created without the FOREIGN KEY constraints. Temporary tables cannot be referenced in FOREIGN KEY constraints.

If a temporary table is created with a named constraint and the temporary table is created within the scope of a user-defined transaction, only one user at a time can execute the statement that creates the temp table. For example, if a stored procedure creates a temporary table with a named primary key constraint, the stored procedure cannot be executed simultaneously by multiple users.

Partitioned Tables

Before creating a partitioned table by using CREATE TABLE, you must first create a partition function to specify how the table becomes partitioned. A partition function is created by using CREATE PARTITION FUNCTION. Second, you must create a partition scheme to specify the filegroups that will hold the partitions indicated by the partition function. A partition scheme is created by using CREATE PARTITION SCHEME. Placement of PRIMARY KEY or UNIQUE constraints to separate filegroups cannot be specified for partitioned tables. For more information, see Partitioned Tables and Indexes.

PRIMARY KEY Constraints

- A table can contain only one PRIMARY KEY constraint.
- The index generated by a PRIMARY KEY constraint cannot cause the number of indexes on the table to exceed 999 nonclustered indexes and 1 clustered index.
- If CLUSTERED or NONCLUSTERED is not specified for a PRIMARY KEY constraint, CLUSTERED is used if there are no clustered indexes specified for UNIQUE constraints.
- All columns defined within a PRIMARY KEY constraint must be defined as NOT NULL. If nullability is not specified, all columns participating in a PRIMARY KEY constraint have their nullability set to NOT NULL.
- If a primary key is defined on a CLR user-defined type column, the implementation of the type must support binary ordering. For more information, see CLR User-defined Types.

UNIQUE Constraints

- If CLUSTERED or NONCLUSTERED is not specified for a UNIQUE constraint, NONCLUSTERED is used by default.
- Each UNIQUE constraint generates an index. The number of UNIQUE constraints cannot cause the number of indexes on the table to exceed 999 nonclustered indexes and 1 clustered index.
- If a unique constraint is defined on a CLR user-defined type column, the implementation of the type must support binary or operator-based ordering. For more information, see CLR User-defined Types.

FOREIGN KEY Constraints

- When a value other than NULL is entered into the column of a FOREIGN KEY constraint, the value must exist in the referenced column; otherwise, a foreign key violation error message is returned.
- FOREIGN KEY constraints are applied to the preceding column, unless source columns are specified.
- FOREIGN KEY constraints can reference only tables within the same database on the same server. Cross-database referential integrity must be implemented through triggers. For more information, see CREATE TRIGGER.
- FOREIGN KEY constraints can reference another column in the same table. This is referred to as a self-reference.
- The REFERENCES clause of a column-level FOREIGN KEY constraint can list only one reference column. This column must have the same data type as the column on which the constraint is defined.
- The REFERENCES clause of a table-level FOREIGN KEY constraint must have the same number of reference columns as the number of columns in the constraint column list. The data type of each reference column must also be the same as the corresponding column in the column list.
- CASCADE, SET NULL or SET DEFAULT cannot be specified if a column of type **timestamp** is part of either the foreign key or the referenced key.
- CASCADE, SET NULL, SET DEFAULT and NO ACTION can be combined on tables that have referential relationships with each other. If the Database Engine encounters NO ACTION, it stops and rolls back related CASCADE, SET NULL and SET DEFAULT actions. When a DELETE statement causes a combination of CASCADE, SET NULL, SET DEFAULT and NO ACTION actions, all the CASCADE, SET NULL and SET DEFAULT actions are applied before the Database Engine checks for any NO ACTION.
- The Database Engine does not have a predefined limit on either the number of FOREIGN KEY constraints a table can contain that reference other tables, or the number of FOREIGN KEY constraints that are owned by other tables that reference a specific table.

Nevertheless, the actual number of FOREIGN KEY constraints that can be used is limited by the hardware configuration and by the design of the database and application. We recommend that a table contain no more than 253 FOREIGN KEY constraints, and that it be referenced by no more than 253 FOREIGN KEY constraints. The effective limit for you may be more or less depending on the application and hardware. Consider the cost of enforcing FOREIGN KEY constraints when you design your database and applications.

- FOREIGN KEY constraints are not enforced on temporary tables.
- FOREIGN KEY constraints can reference only columns in PRIMARY KEY or UNIQUE constraints in the referenced table or in a UNIQUE INDEX on the referenced table.
- If a foreign key is defined on a CLR user-defined type column, the implementation of the type must support binary ordering. For more information, see CLR User-defined Types.
- A column of type **varchar(max)** can participate in a FOREIGN KEY constraint only if the primary key it references is also defined as type **varchar(max)**.

DEFAULT Definitions

- A column can have only one DEFAULT definition.
- A DEFAULT definition can contain constant values, functions, SQL-92 niladic functions, or NULL. The following table shows the niladic functions and the values they return for the default during an INSERT statement.

SQL-92 niladic function	Value returned
CURRENT_TIMESTAMP	Current date and time.
CURRENT_USER	Name of user performing an insert.
SESSION_USER	Name of user performing an insert.
SYSTEM_USER	Name of user performing an insert.
USER	Name of user performing an insert.

- constant_expression in a DEFAULT definition cannot refer to another column in the table, or to other tables, views, or stored procedures.
- DEFAULT definitions cannot be created on columns with a **timestamp** data type or columns with an IDENTITY property.
- DEFAULT definitions cannot be created for columns with alias data types if the alias data type is bound to a default object.

CHECK Constraints

- A column can have any number of CHECK constraints, and the condition can include multiple logical expressions combined with AND and OR. Multiple CHECK constraints for a column are validated in the order they are created.

- The search condition must evaluate to a Boolean expression and cannot reference another table.
- A column-level CHECK constraint can reference only the constrained column, and a table-level CHECK constraint can reference only columns in the same table.

  CHECK CONSTRAINTS and rules serve the same function of validating the data during INSERT and UPDATE statements.
- When a rule and one or more CHECK constraints exist for a column or columns, all restrictions are evaluated.
- CHECK constraints cannot be defined on **text**, **ntext**, or **image** columns.

Additional Constraint Information

- An index created for a constraint cannot be dropped by using DROP INDEX; the constraint must be dropped by using ALTER TABLE. An index created for and used by a constraint can be rebuilt by using DBCC DBREINDEX.
- Constraint names must follow the rules for identifiers, except that the name cannot start with a number sign (#). If constraint_name is not supplied, a system-generated name is assigned to the constraint. The constraint name appears in any error message about constraint violations.
- When a constraint is violated in an INSERT, UPDATE, or DELETE statement, the statement is ended. However, when SET XACT_ABORT is set to OFF, the transaction, if the statement is part of an explicit transaction, continues to be processed. When SET XACT_ABORT is set to ON, the whole transaction is rolled back. You can also use the ROLLBACK TRANSACTION statement with the transaction definition by checking the **@@**ERROR system function.
- When ALLOW_ROW_LOCKS = ON and ALLOW_PAGE_LOCK = ON, row-, page-, and table-level locks are allowed when you access the index. The Database Engine chooses the appropriate lock and can escalate the lock from a row or page lock to a table lock. When ALLOW_ROW_LOCKS = OFF and ALLOW_PAGE_LOCK = OFF, only a table-level lock is allowed when you access the index.
- If a table has FOREIGN KEY or CHECK CONSTRAINTS and triggers, the constraint conditions are evaluated before the trigger is executed.

For a report on a table and its columns, use **sp_help** or **sp_helpconstraint**. To rename a table, use **sp_rename**. For a report on the views and stored procedures that depend on a table, use sys.dm_sql_referenced_entities and sys.dm_sql_referencing_entities.

Nullability Rules Within a Table Definition

The nullability of a column determines whether that column can allow a null value (NULL) as the data in that column. NULL is not zero or blank: NULL means no entry was made or an explicit NULL was supplied, and it typically implies that the value is either unknown or not applicable.

When you use CREATE TABLE or ALTER TABLE to create or alter a table, database and session settings influence and possibly override the nullability of the data type that is used in a column definition. We recommend that you always explicitly define a column as NULL or NOT NULL for

noncomputed columns or, if you use a user-defined data type, that you allow the column to use the default nullability of the data type. Sparse columns must always allow NULL.

When column nullability is not explicitly specified, column nullability follows the rules shown in the following table.

Column data type	Rule
Alias data type	The Database Engine uses the nullability that is specified when the data type was created. To determine the default nullability of the data type, use **sp_help**.
CLR user-defined type	Nullability is determined according to the column definition.
System-supplied data type	If the system-supplied data type has only one option, it takes precedence. **timestamp** data types must be NOT NULL. When any session settings are set ON by using SET: • ANSI_NULL_DFLT_ON = ON, NULL is assigned. • ANSI_NULL_DFLT_OFF = ON, NOT NULL is assigned. • When any database settings are configured by using ALTER DATABASE: • ANSI_NULL_DEFAULT_ON = ON, NULL is assigned. • ANSI_NULL_DEFAULT_OFF = ON, NOT NULL is assigned. • To view the database setting for ANSI_NULL_DEFAULT, use the **sys.databases** catalog view

When neither of the ANSI_NULL_DFLT options is set for the session and the database is set to the default (ANSI_NULL_DEFAULTis OFF), the default of NOT NULL is assigned.

If the column is a computed column, its nullability is always automatically determined by the Database Engine. To find out the nullability of this type of column, use the COLUMNPROPERTY function with the **AllowsNull** property.

Note

The SQL Server ODBC driver and Microsoft OLE DB Provider for SQL Server both default to having ANSI_NULL_DFLT_ON set to ON. ODBC and OLE DB users can configure this in ODBC data sources, or with connection attributes or properties set by the application.

Data Compression

System tables cannot be enabled for compression. When you are creating a table, data compression is set to NONE, unless specified otherwise. If you specify a list of partitions or a partition that is out of range, an error will be generated. For a more information about data compression, see Creating Compressed Tables and Indexes.

To evaluate how changing the compression state will affect a table, an index, or a partition, use the sp_estimate_data_compression_savings stored procedure.

Permissions

Requires CREATE TABLE permission in the database and ALTER permission on the schema in which the table is being created.

If any columns in the CREATE TABLE statement are defined to be of a CLR user-defined type, either ownership of the type or REFERENCES permission on it is required.

If any columns in the CREATE TABLE statement have an XML schema collection associated with them, either ownership of the XML schema collection or REFERENCES permission on it is required.

Any user can create temporary tables in tempdb.

Examples

A. Using PRIMARY KEY constraints

The following example shows the column definition for a PRIMARY KEY constraint with a clustered index on the `EmployeeID` column of the `Employee` table (allowing the system to supply the constraint name) in the `AdventureWorks` sample database.

```
EmployeeID int
PRIMARY KEY CLUSTERED
```

B. Using FOREIGN KEY constraints

A FOREIGN KEY constraint is used to reference another table. Foreign keys can be single-column keys or multicolumn keys. This following example shows a single-column FOREIGN KEY constraint on the `SalesOrderHeader` table that references the `SalesPerson` table. Only the REFERENCES clause is required for a single-column FOREIGN KEY constraint.

```
SalesPersonID int NULL
REFERENCES SalesPerson(SalesPersonID)
```

You can also explicitly use the FOREIGN KEY clause and restate the column attribute. Note that the column name does not have to be the same in both tables.

```
FOREIGN KEY (SalesPersonID) REFERENCES SalesPerson(SalesPersonID)
```

Multicolumn key constraints are created as table constraints. In the database, the `SpecialOfferProduct` table includes a multicolumn PRIMARY KEY. The following example shows how to reference this key from another table; an explicit constraint name is optional.

```
CONSTRAINT FK_SpecialOfferProduct_SalesOrderDetail FOREIGN KEY
 (ProductID, SpecialOfferID)
REFERENCES SpecialOfferProduct (ProductID, SpecialOfferID)
```

C. Using UNIQUE constraints

UNIQUE constraints are used to enforce uniqueness on nonprimary key columns. The following example enforces a restriction that the `Name` column of the `Product` table must be unique.

```
Name nvarchar(100) NOT NULL
UNIQUE NONCLUSTERED
```

D. Using DEFAULT definitions

Defaults supply a value (with the INSERT and UPDATE statements) when no value is supplied. For example, the database could include a lookup table listing the different jobs employees can fill in the company. Under a column that describes each job, a character string default could supply a description when an actual description is not entered explicitly.

```
DEFAULT 'New Position - title not formalized yet'
```

In addition to constants, DEFAULT definitions can include functions. Use the following example to get the current date for an entry.

```
DEFAULT (getdate())
```

A niladic-function scan can also improve data integrity. To keep track of the user that inserted a row, use the niladic-function for USER. Do not enclose the niladic-functions with parentheses.

```
DEFAULT USER
```

E. Using CHECK constraints

The following example shows a restriction made to values that are entered into the `CreditRating` column of the `Vendor` table. The constraint is unnamed.

```
CHECK (CreditRating >= 1 and CreditRating <= 5)
```

This example shows a named constraint with a pattern restriction on the character data entered into a column of a table.

```
CONSTRAINT CK_emp_id CHECK (emp_id LIKE
'[A-Z][A-Z][A-Z][1-9][0-9][0-9][0-9][0-9][FM]'
OR emp_id LIKE '[A-Z]-[A-Z][1-9][0-9][0-9][0-9][0-9][FM]')
```

This example specifies that the values must be within a specific list or follow a specified pattern.

```
CHECK (emp_id IN ('1389', '0736', '0877', '1622', '1756')
OR emp_id LIKE '99[0-9][0-9]')
```

F. Showing the complete table definition

The following example shows the complete table definitions with all constraint definitions for table `PurchaseOrderDetail` created in the database. Note that to run the sample, the table schema is changed to `dbo`.

```
CREATE TABLE dbo.PurchaseOrderDetail
(
    PurchaseOrderID int NOT NULL
        REFERENCES Purchasing.PurchaseOrderHeader(PurchaseOrderID),
    LineNumber smallint NOT NULL,
    ProductID int NULL
        REFERENCES Production.Product(ProductID),
    UnitPrice money NULL,
    OrderQty smallint NULL,
    ReceivedQty float NULL,
    RejectedQty float NULL,
    DueDate datetime NULL,
    rowguid uniqueidentifier ROWGUIDCOL  NOT NULL
        CONSTRAINT DF_PurchaseOrderDetail_rowguid DEFAULT (newid()),
    ModifiedDate datetime NOT NULL
        CONSTRAINT DF_PurchaseOrderDetail_ModifiedDate DEFAULT (getdate()),
    LineTotal  AS ((UnitPrice*OrderQty)),
    StockedQty  AS ((ReceivedQty-RejectedQty)),
    CONSTRAINT PK_PurchaseOrderDetail_PurchaseOrderID_LineNumber
               PRIMARY KEY CLUSTERED (PurchaseOrderID, LineNumber)
               WITH (IGNORE_DUP_KEY = OFF)
)
ON PRIMARY;
```

G. Creating a table with an xml column typed to an XML schema collection

The following example creates a table with an `xml` column that is typed to XML schema collection `HRResumeSchemaCollection`. The `DOCUMENT` keyword specifies that each instance of the `xml` data type in column_name can contain only one top-level element.

```
USE AdventureWorks2012;
GO
CREATE TABLE HumanResources.EmployeeResumes
   (LName nvarchar(25), FName nvarchar(25),
    Resume xml( DOCUMENT HumanResources.HRResumeSchemaCollection) );
```

H. Creating a partitioned table

The following example creates a partition function to partition a table or index into four partitions. Then, the example creates a partition scheme that specifies the filegroups in which to hold each of the four partitions. Finally, the example creates a table that uses the partition scheme. This example assumes the filegroups already exist in the database.

```
CREATE PARTITION FUNCTION myRangePF1 (int)
    AS RANGE LEFT FOR VALUES (1, 100, 1000) ;
GO

CREATE PARTITION SCHEME myRangePS1
    AS PARTITION myRangePF1
    TO (test1fg, test2fg, test3fg, test4fg) ;
GO

CREATE TABLE PartitionTable (col1 int, col2 char(10))
    ON myRangePS1 (col1) ;
GO
```

Based on the values of column `col1` of `PartitionTable`, the partitions are assigned in the following ways.

Filegroup	test1fg	test2fg	test3fg	test4fg
Partition	1	2	3	4
Values	col 1 <= 1	col1 > 1 AND col1 <= 100	col1 > 100 AND col1 <= 1,000	col1 > 1000

I. Using the uniqueidentifier data type in a column

The following example creates a table with a `uniqueidentifier` column. The example uses a PRIMARY KEY constraint to protect the table against users inserting duplicated values, and it uses the `NEWSEQUENTIALID()` function in the `DEFAULT` constraint to provide values for new rows. The ROWGUIDCOL property is applied to the `uniqueidentifier` column so that it can be referenced using the $ROWGUID keyword.

```
CREATE TABLE dbo.Globally_Unique_Data
    (guid uniqueidentifier CONSTRAINT Guid_Default DEFAULT NEWSEQUENTIALID()
ROWGUIDCOL,
    Employee_Name varchar(60)
    CONSTRAINT Guid_PK PRIMARY KEY (guid) );
```

J. Using an expression for a computed column

The following example shows the use of an expression (`(low + high)/2`) for calculating the `myavg` computed column.

```
CREATE TABLE dbo.mytable
    ( low int, high int, myavg AS (low + high)/2 ) ;
```

K. Creating a computed column based on a user-defined type column

The following example creates a table with one column defined as user-defined type `utf8string`, assuming that the type's assembly, and the type itself, have already been created in the current database. A second column is defined based on `utf8string`, and uses method `ToString()` of **type(class)** `utf8string` to compute a value for the column.

```
CREATE TABLE UDTypeTable
    ( u utf8string, ustr AS u.ToString() PERSISTED ) ;
```

L. Using the USER_NAME function for a computed column

The following example uses the `USER_NAME()` function in the `myuser_name` column.

```
CREATE TABLE dbo.mylogintable
    ( date_in datetime, user_id int, myuser_name AS USER_NAME() ) ;
```

M. Creating a table that has a FILESTREAM column

The following example creates a table that has a `FILESTREAM` column `Photo`. If a table has one or more `FILESTREAM` columns, the table must have one `ROWGUIDCOL` column.

```
CREATE TABLE dbo.EmployeePhoto
    (
    EmployeeId int NOT NULL PRIMARY KEY
    ,Photo varbinary(max) FILESTREAM NULL
    ,MyRowGuidColumn uniqueidentifier NOT NULL ROWGUIDCOL
        UNIQUE DEFAULT NEWID()
    );
```

N. Creating a table that uses row compression

The following example creates a table that uses row compression.

```
CREATE TABLE dbo.T1
(c1 int, c2 nvarchar(200) )
WITH (DATA_COMPRESSION = ROW);
```

For additional data compression examples, see Creating Compressed Tables and Indexes.

O. Creating a table that has sparse columns and a column set

The following examples show to how to create a table that has a sparse column, and a table that has two sparse columns and a column set. The examples use the basic syntax. For more complex examples, see Using Sparse Columns and Using Sparse Column Sets.

This example creates a table that has a sparse column.

```
CREATE TABLE dbo.T1
    (c1 int PRIMARY KEY,
    c2 varchar(50) SPARSE NULL ) ;
```

This example creates a table that has two sparse columns and a column set named `CSet`.

```
CREATE TABLE T1
    (c1 int PRIMARY KEY,
    c2 varchar(50) SPARSE NULL,
    c3 int SPARSE NULL,
    CSet XML COLUMN_SET FOR ALL_SPARSE_COLUMNS ) ;
```

See Also

ALTER TABLE
COLUMNPROPERTY
CREATE INDEX
CREATE VIEW
Data Types
DROP INDEX
sys.dm_sql_referenced_entities
sys.dm_sql_referencing_entities
DROP TABLE
CREATE PARTITION FUNCTION
CREATE PARTITION SCHEME
CREATE TYPE
EVENTDATA
sp_help
sp_helpconstraint
sp_rename
sp_spaceused

IDENTITY (Property)

Creates an identity column in a table. This property is used with the CREATE TABLE and ALTER TABLE Transact-SQL statements.

Note
The IDENTITY property is different from the SQL-DMO **Identity** property that exposes the row identity property of a column.

Transact-SQL Syntax Conventions

Syntax

IDENTITY [**(seed , increment)**]

Arguments

seed

Is the value that is used for the very first row loaded into the table.

increment

Is the incremental value that is added to the identity value of the previous row that was loaded.

You must specify both the seed and increment or neither. If neither is specified, the default is (1,1).

Remarks

If an identity column exists for a table with frequent deletions, gaps can occur between identity values. If this is a concern, do not use the IDENTITY property. However, to make sure that no gaps have been created or to fill an existing gap, evaluate the existing identity values before explicitly entering one with SET IDENTITY_INSERT ON.

If you are reusing a removed identity value, use the sample code in Example B to look for the next available identity value. Replace `tablename`, `column_type`, and `MAX(column_type) - 1` with a table name, identity column data type, and numeric value of the maximum allowed value (for that data type) -1.

Use DBCC CHECKIDENT to check the current identity value and compare it with the maximum value in the identity column.

If a table with an identity column is published for replication, the identity column must be managed in a way that is appropriate for the type of replication used. For more information, see Replicating Identity Columns.

Note
To create an automatically incrementing number that can be used in multiple tables or that can be called from applications without referencing any table, see Creating and Using Sequence Numbers.

Examples

A. Using the IDENTITY property with CREATE TABLE

The following example creates a new table using the `IDENTITY` property for an automatically incrementing identification number.

```
USE AdventureWorks2012
IF OBJECT_ID ('dbo.new_employees', 'U') IS NOT NULL
   DROP TABLE new_employees;
GO
CREATE TABLE new_employees
(
 id_num int IDENTITY(1,1),
 fname varchar (20),
 minit char(1),
 lname varchar(30)
);

INSERT new_employees
   (fname, minit, lname)
VALUES
   ('Karin', 'F', 'Josephs');

INSERT new_employees
   (fname, minit, lname)
VALUES
   ('Pirkko', 'O', 'Koskitalo');
```

B. Using generic syntax for finding gaps in identity values

The following example shows generic syntax for finding gaps in identity values when data is removed.

Note

The first part of the following Transact-SQL script is designed for illustration only. You can run the Transact-SQL script that starts with the comment: `-- Create the img table`.

```
-- Here is the generic syntax for finding identity value gaps in data.
-- The illustrative example starts here.
```

```
SET IDENTITY_INSERT tablename ON
DECLARE @minidentval column_type
DECLARE @maxidentval column_type
DECLARE @nextidentval column_type
SELECT @minidentval = MIN($IDENTITY), @maxidentval = MAX($IDENTITY)
    FROM tablename
IF @minidentval = IDENT_SEED('tablename')
   SELECT @nextidentval = MIN($IDENTITY) + IDENT_INCR('tablename')
   FROM tablename t1
   WHERE $IDENTITY BETWEEN IDENT_SEED('tablename') AND
      @maxidentval AND
      NOT EXISTS (SELECT * FROM tablename t2
         WHERE t2.$IDENTITY = t1.$IDENTITY +
            IDENT_INCR('tablename'))
ELSE
   SELECT @nextidentval = IDENT_SEED('tablename')
SET IDENTITY_INSERT tablename OFF
-- Here is an example to find gaps in the actual data.
-- The table is called img and has two columns: the first column
-- called id_num, which is an increasing identification number, and the
-- second column called company_name.
-- This is the end of the illustration example.

-- Create the img table.
-- If the img table already exists, drop it.
-- Create the img table.
IF OBJECT_ID ('dbo.img', 'U') IS NOT NULL
   DROP TABLE img
GO
CREATE TABLE img (id_num int IDENTITY(1,1), company_name sysname)
INSERT img(company_name) VALUES ('New Moon Books')
INSERT img(company_name) VALUES ('Lucerne Publishing')
-- SET IDENTITY_INSERT ON and use in img table.
```

```
SET IDENTITY_INSERT img ON

DECLARE @minidentval smallint
DECLARE @nextidentval smallint
SELECT @minidentval = MIN($IDENTITY) FROM img
 IF @minidentval = IDENT_SEED('img')
    SELECT @nextidentval = MIN($IDENTITY) + IDENT_INCR('img')
    FROM img t1
    WHERE $IDENTITY BETWEEN IDENT_SEED('img') AND 32766 AND
      NOT    EXISTS (SELECT * FROM img t2
           WHERE t2.$IDENTITY = t1.$IDENTITY + IDENT_INCR('img'))
 ELSE
    SELECT @nextidentval = IDENT_SEED('img')
SET IDENTITY_INSERT img OFF
```

See Also

ALTER TABLE
CREATE TABLE
DBCC CHECKIDENT
IDENT_INCR
@@IDENTITY
IDENTITY (Function)
IDENT_SEED
SELECT
SET IDENTITY_INSERT
Replicating Identity Columns

CREATE TRIGGER

Creates a DML, DDL, or logon trigger in SQL Server 2012. A trigger is a special kind of stored procedure that automatically executes when an event occurs in the database server. DML triggers execute when a user tries to modify data through a data manipulation language (DML) event. DML events are INSERT, UPDATE, or DELETE statements on a table or view. These triggers fire when any valid event is fired, regardless of whether or not any table rows are affected. For more information, see DML Triggers.

DDL triggers execute in response to a variety of data definition language (DDL) events. These events primarily correspond to Transact-SQL CREATE, ALTER, and DROP statements, and certain

system stored procedures that perform DDL-like operations. Logon triggers fire in response to the LOGON event that is raised when a user sessions is being established. Triggers can be created directly from Transact-SQL statements or from methods of assemblies that are created in the Microsoft .NET Framework common language runtime (CLR) and uploaded to an instance of SQL Server. SQL Server allows for creating multiple triggers for any specific statement.

noteDXDOC112778PADS Security Note

Malicious code inside triggers can run under escalated privileges. For more information on how to mitigate this threat, see Using Large-Value Data Types.

Transact-SQL Syntax Conventions

Syntax

Trigger on an INSERT, UPDATE, or DELETE statement to a table or view (DML Trigger)

```
CREATE TRIGGER [ schema_name . ]trigger_name
ON { table | view }
[ WITH <dml_trigger_option> [ ,...n ] ]
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
[ NOT FOR REPLICATION ]
AS { sql_statement [ ; ] [ ,...n ] | EXTERNAL NAME <method specifier [ ; ] > }
```

<dml_trigger_option> ::=

```
[ ENCRYPTION ]
[ EXECUTE AS Clause ]
```

<method_specifier> ::=

```
assembly_name.class_name.method_name
```

Trigger on a CREATE, ALTER, DROP, GRANT, DENY, REVOKE, or UPDATE STATISTICS statement (DDL Trigger)

```
CREATE TRIGGER trigger_name
ON { ALL SERVER | DATABASE }
[ WITH <ddl_trigger_option> [ ,...n ] ]
{ FOR | AFTER } { event_type | event_group } [ ,...n ]
AS { sql_statement [ ; ] [ ,...n ] | EXTERNAL NAME < method specifier > [ ; ] }
```

<ddl_trigger_option> ::=

[ENCRYPTION]

[**EXECUTE AS Clause**]

Trigger on a LOGON event (Logon Trigger)

CREATE TRIGGER **trigger_name**

ON ALL SERVER

[WITH <logon_trigger_option> [**,...n**]]

{ FOR | AFTER } LOGON

AS { **sql_statement** [;] [**,...n**] | EXTERNAL NAME < method specifier > [;] }

<logon_trigger_option> ::=

[ENCRYPTION]

[**EXECUTE AS Clause**]

Arguments

schema_name

Is the name of the schema to which a DML trigger belongs. DML triggers are scoped to the schema of the table or view on which they are created. schema_name cannot be specified for DDL or logon triggers.

trigger_name

Is the name of the trigger. A trigger_name must comply with the rules for identifiers, except that trigger_name cannot start with # or ##.

table | view

Is the table or view on which the DML trigger is executed and is sometimes referred to as the trigger table or trigger view. Specifying the fully qualified name of the table or view is optional. A view can be referenced only by an INSTEAD OF trigger. DML triggers cannot be defined on local or global temporary tables.

DATABASE

Applies the scope of a DDL trigger to the current database. If specified, the trigger fires whenever event_type or event_group occurs in the current database.

ALL SERVER

Applies the scope of a DDL or logon trigger to the current server. If specified, the trigger fires whenever event_type or event_group occurs anywhere in the current server.

WITH ENCRYPTION

Obfuscates the text of the CREATE TRIGGER statement. Using WITH ENCRYPTION prevents the trigger from being published as part of SQL Server replication. WITH ENCRYPTION

cannot be specified for CLR triggers.

EXECUTE AS

Specifies the security context under which the trigger is executed. Enables you to control which user account the instance of SQL Server uses to validate permissions on any database objects that are referenced by the trigger.

For more information, see EXECUTE AS.

FOR | AFTER

AFTER specifies that the DML trigger is fired only when all operations specified in the triggering SQL statement have executed successfully. All referential cascade actions and constraint checks also must succeed before this trigger fires.

AFTER is the default when FOR is the only keyword specified.

AFTER triggers cannot be defined on views.

INSTEAD OF

Specifies that the DML trigger is executed *instead of* the triggering SQL statement, therefore, overriding the actions of the triggering statements. INSTEAD OF cannot be specified for DDL or logon triggers.

At most, one INSTEAD OF trigger per INSERT, UPDATE, or DELETE statement can be defined on a table or view. However, you can define views on views where each view has its own INSTEAD OF trigger.

INSTEAD OF triggers are not allowed on updatable views that use WITH CHECK OPTION. SQL Server raises an error when an INSTEAD OF trigger is added to an updatable view WITH CHECK OPTION specified. The user must remove that option by using ALTER VIEW before defining the INSTEAD OF trigger.

{ [DELETE] [,] [INSERT] [,] [UPDATE] }

Specifies the data modification statements that activate the DML trigger when it is tried against this table or view. At least one option must be specified. Any combination of these options in any order is allowed in the trigger definition.

For INSTEAD OF triggers, the DELETE option is not allowed on tables that have a referential relationship specifying a cascade action ON DELETE. Similarly, the UPDATE option is not allowed on tables that have a referential relationship specifying a cascade action ON UPDATE.

event_type

Is the name of a Transact-SQL language event that, after execution, causes a DDL trigger to fire. Valid events for DDL triggers are listed in DDL Events.

event_group

Is the name of a predefined grouping of Transact-SQL language events. The DDL trigger fires after execution of any Transact-SQL language event that belongs to event_group. Valid event

groups for DDL triggers are listed in DDL Event Groups.

After the CREATE TRIGGER has finished running, event_group also acts as a macro by adding the event types it covers to the sys.trigger_events catalog view.

NOT FOR REPLICATION

Indicates that the trigger should not be executed when a replication agent modifies the table that is involved in the trigger.

sql_statement

Is the trigger conditions and actions. Trigger conditions specify additional criteria that determine whether the tried DML, DDL, or logon events cause the trigger actions to be performed.

The trigger actions specified in the Transact-SQL statements go into effect when the operation is tried.

Triggers can include any number and kind of Transact-SQL statements, with exceptions. For more information, see Remarks. A trigger is designed to check or change data based on a data modification or definition statement; it should not return data to the user. The Transact-SQL statements in a trigger frequently include control-of-flow language.

DML triggers use the deleted and inserted logical (conceptual) tables. They are structurally similar to the table on which the trigger is defined, that is, the table on which the user action is tried. The deleted and inserted tables hold the old values or new values of the rows that may be changed by the user action. For example, to retrieve all values in the `deleted` table, use:

```
SELECT * FROM deleted
```

For more information, see Using the inserted and deleted Tables.

DDL and logon triggers capture information about the triggering event by using the eventdata (Transact-SQL) function. For more information, see Using the eventdata Function.

SQL Server allows for the update of **text**, **ntext**, or **image** columns through the INSTEAD OF trigger on tables or views.

Important

ntext, **text**, and **image** data types will be removed in a future version of Microsoft SQL Server. Avoid using these data types in new development work, and plan to modify applications that currently use them. Use nvarchar(max), varchar(max), and varbinary(max) instead. Both AFTER and INSTEAD OF triggers support **varchar(MAX)**, **nvarchar(MAX)**, and **varbinary(MAX)** data in the inserted and deleted tables.

< method_specifier >

For a CLR trigger, specifies the method of an assembly to bind with the trigger. The method must take no arguments and return void. class_name must be a valid SQL Server identifier and must exist as a class in the assembly with assembly visibility. If the class has a

namespace-qualified name that uses '.' to separate namespace parts, the class name must be delimited by using [ ] or " " delimiters. The class cannot be a nested class.

Note

By default, the ability of SQL Server to run CLR code is off. You can create, modify, and drop database objects that reference managed code modules, but these references will not execute in an instance of SQL Server unless the clr enabled Option is enabled by using sp_configure.

Remarks

DML Triggers

DML triggers are frequently used for enforcing business rules and data integrity. SQL Server provides declarative referential integrity (DRI) through the ALTER TABLE and CREATE TABLE statements. However, DRI does not provide cross-database referential integrity. Referential integrity refers to the rules about the relationships between the primary and foreign keys of tables. To enforce referential integrity, use the PRIMARY KEY and FOREIGN KEY constraints in ALTER TABLE and CREATE TABLE. If constraints exist on the trigger table, they are checked after the INSTEAD OF trigger execution and before the AFTER trigger execution. If the constraints are violated, the INSTEAD OF trigger actions are rolled back and the AFTER trigger is not fired.

The first and last AFTER triggers to be executed on a table can be specified by using sp_settriggerorder. Only one first and one last AFTER trigger for each INSERT, UPDATE, and DELETE operation can be specified on a table. If there are other AFTER triggers on the same table, they are randomly executed.

If an ALTER TRIGGER statement changes a first or last trigger, the first or last attribute set on the modified trigger is dropped, and the order value must be reset by using sp_settriggerorder.

An AFTER trigger is executed only after the triggering SQL statement has executed successfully. This successful execution includes all referential cascade actions and constraint checks associated with the object updated or deleted.

If an INSTEAD OF trigger defined on a table executes a statement against the table that would ordinarily fire the INSTEAD OF trigger again, the trigger is not called recursively. Instead, the statement is processed as if the table had no INSTEAD OF trigger and starts the chain of constraint operations and AFTER trigger executions. For example, if a trigger is defined as an INSTEAD OF INSERT trigger for a table, and the trigger executes an INSERT statement on the same table, the INSERT statement executed by the INSTEAD OF trigger does not call the trigger again. The INSERT executed by the trigger starts the process of performing constraint actions and firing any AFTER INSERT triggers defined for the table.

If an INSTEAD OF trigger defined on a view executes a statement against the view that would ordinarily fire the INSTEAD OF trigger again, it is not called recursively. Instead, the statement is resolved as modifications against the base tables underlying the view. In this case, the view definition must meet all the restrictions for an updatable view. For a definition of updatable views, see Modifying Data Through a View.

For example, if a trigger is defined as an INSTEAD OF UPDATE trigger for a view, and the trigger executes an UPDATE statement referencing the same view, the UPDATE statement executed by the INSTEAD OF trigger does not call the trigger again. The UPDATE executed by the trigger is processed against the view as if the view did not have an INSTEAD OF trigger. The columns changed by the UPDATE must be resolved to a single base table. Each modification to an underlying base table starts the chain of applying constraints and firing AFTER triggers defined for the table.

Testing for UPDATE or INSERT Actions to Specific Columns

You can design a Transact-SQL trigger to perform certain actions based on UPDATE or INSERT modifications to specific columns. Use UPDATE() or COLUMNS_UPDATED in the body of the trigger for this purpose. UPDATE() tests for UPDATE or INSERT tries on one column. COLUMNS_UPDATED tests for UPDATE or INSERT actions that are performed on multiple columns and returns a bit pattern that indicates which columns were inserted or updated.

Trigger Limitations

CREATE TRIGGER must be the first statement in the batch and can apply to only one table.

A trigger is created only in the current database; however, a trigger can reference objects outside the current database.

If the trigger schema name is specified to qualify the trigger, qualify the table name in the same way.

The same trigger action can be defined for more than one user action (for example, INSERT and UPDATE) in the same CREATE TRIGGER statement.

INSTEAD OF DELETE/UPDATE triggers cannot be defined on a table that has a foreign key with a cascade on DELETE/UPDATE action defined.

Any SET statement can be specified inside a trigger. The SET option selected remains in effect during the execution of the trigger and then reverts to its former setting.

When a trigger fires, results are returned to the calling application, just like with stored procedures. To prevent having results returned to an application because of a trigger firing, do not include either SELECT statements that return results or statements that perform variable assignment in a trigger. A trigger that includes either SELECT statements that return results to the user or statements that perform variable assignment requires special handling; these returned results would have to be written into every application in which modifications to the trigger table are allowed. If variable assignment must occur in a trigger, use a SET NOCOUNT statement at the start of the trigger to prevent the return of any result sets.

Although a TRUNCATE TABLE statement is in effect a DELETE statement, it does not activate a trigger because the operation does not log individual row deletions. However, only those users with permissions to execute a TRUNCATE TABLE statement need be concerned about inadvertently circumventing a DELETE trigger this way.

The WRITETEXT statement, whether logged or unlogged, does not activate a trigger.

The following Transact-SQL statements are not allowed in a DML trigger:

ALTER DATABASE	CREATE DATABASE	DROP DATABASE
RESTORE DATABASE	RESTORE LOG	RECONFIGURE

Additionally, the following Transact-SQL statements are not allowed inside the body of a DML trigger when it is used against the table or view that is the target of the triggering action.

CREATE INDEX (including CREATE SPATIAL INDEX and CREATE XML INDEX)	ALTER INDEX	DROP INDEX
DBCC DBREINDEX	ALTER PARTITION FUNCTION	DROP TABLE
ALTER TABLE when used to do the following: • Add, modify, or drop columns. • Switch partitions. • Add or drop PRIMARY KEY or UNIQUE constraints.		

Note
Because SQL Server does not support user-defined triggers on system tables, we recommend that you do not create user-defined triggers on system tables.

DDL Triggers

DDL triggers, like standard triggers, execute stored procedures in response to an event. But unlike standard triggers, they do not execute in response to UPDATE, INSERT, or DELETE statements on a table or view. Instead, they primarily execute in response to data definition language (DDL) statements. These include CREATE, ALTER, DROP, GRANT, DENY, REVOKE, and UPDATE STATISTICS statements. Certain system stored procedures that perform DDL-like operations can also fire DDL triggers.

Important
Test your DDL triggers to determine their responses to system stored procedure execution. For example, the CREATE TYPE statement and the sp_addtype and sp_rename stored procedures will fire a DDL trigger that is created on a CREATE_TYPE event.

For more information about DDL triggers, see DDL Triggers.

DDL triggers do not fire in response to events that affect local or global temporary tables and stored procedures.

Unlike DML triggers, DDL triggers are not scoped to schemas. Therefore, functions such as OBJECT_ID, OBJECT_NAME, OBJECTPROPERTY, and OBJECTPROPERTYEX cannot be used for querying metadata about DDL triggers. Use the catalog views instead. For more information, see Getting Information about DDL Triggers.

Note
Server-scoped DDL triggers appear in the SQL Server Management Studio Object Explorer in the **Triggers** folder. This folder is located under the **Server Objects** folder. Database-scoped DDL Triggers appear in the **Database Triggers** folder. This folder is located under the **Programmability** folder of the corresponding database.

Logon Triggers

Logon triggers execute stored procedures in response to a LOGON event. This event is raised when a user session is established with an instance of SQL Server. Logon triggers fire after the authentication phase of logging in finishes, but before the user session is actually established. Therefore, all messages originating inside the trigger that would typically reach the user, such as error messages and messages from the PRINT statement, are diverted to the SQL Server error log. For more information, see Logon Triggers.

Logon triggers do not fire if authentication fails.

Distributed transactions are not supported in a logon trigger. Error 3969 is returned when a logon trigger containing a distributed transaction is fired.

Disabling a Logon Trigger

A logon trigger can effectively prevent successful connections to the Database Engine for all users, including members of the **sysadmin** fixed server role. When a logon trigger is preventing connections, members of the **sysadmin** fixed server role can connect by using the dedicated administrator connection, or by starting the Database Engine in minimal configuration mode (-f). For more information, see Database Engine Service Startup Options.

General Trigger Considerations

Returning Results

The ability to return results from triggers will be removed in a future version of SQL Server. Triggers that return result sets may cause unexpected behavior in applications that are not designed to work with them. Avoid returning result sets from triggers in new development work, and plan to modify applications that currently do this. To prevent triggers from returning result sets, set the disallow results from triggers option to 1.

Logon triggers always disallow results sets to be returned and this behavior is not configurable. If a logon trigger does generate a result set, the trigger fails to execute and the login attempt that fired the trigger is denied.

Multiple Triggers

SQL Server allows for multiple triggers to be created for each DML, DDL, or LOGON event. For example, if CREATE TRIGGER FOR UPDATE is executed for a table that already has an UPDATE

trigger, an additional update trigger is created. In earlier versions of SQL Server, only one trigger for each INSERT, UPDATE, or DELETE data modification event is allowed for each table.

Recursive Triggers

SQL Server also allows for recursive invocation of triggers when the RECURSIVE_TRIGGERS setting is enabled using ALTER DATABASE.

Recursive triggers enable the following types of recursion to occur:

- Indirect recursion

  With indirect recursion, an application updates table T1. This fires trigger TR1, updating table T2. In this scenario, trigger T2 then fires and updates table T1.

- Direct recursion

  With direct recursion, the application updates table T1. This fires trigger TR1, updating table T1. Because table T1 was updated, trigger TR1 fires again, and so on.

The following example uses both indirect and direct trigger recursion Assume that two update triggers, TR1 and TR2, are defined on table T1. Trigger TR1 updates table T1 recursively. An UPDATE statement executes each TR1 and TR2 one time. Additionally, the execution of TR1 triggers the execution of TR1 (recursively) and TR2. The inserted and deleted tables for a specific trigger contain rows that correspond only to the UPDATE statement that invoked the trigger.

Note

The previous behavior occurs only if the RECURSIVE_TRIGGERS setting is enabled by using ALTER DATABASE. There is no defined order in which multiple triggers defined for a specific event are executed. Each trigger should be self-contained.

Disabling the RECURSIVE_TRIGGERS setting only prevents direct recursions. To disable indirect recursion also, set the nested triggers server option to 0 by using sp_configure.

If any one of the triggers performs a ROLLBACK TRANSACTION, regardless of the nesting level, no more triggers are executed.

Nested Triggers

Triggers can be nested to a maximum of 32 levels. If a trigger changes a table on which there is another trigger, the second trigger is activated and can then call a third trigger, and so on. If any trigger in the chain sets off an infinite loop, the nesting level is exceeded and the trigger is canceled. When a Transact-SQL trigger executes managed code by referencing a CLR routine, type, or aggregate, this reference counts as one level against the 32-level nesting limit. Methods invoked from within managed code do not count against this limit

To disable nested triggers, set the nested triggers option of sp_configure to 0 (off). The default configuration allows for nested triggers. If nested triggers is off, recursive triggers is also disabled, regardless of the RECURSIVE_TRIGGERS setting set by using ALTER DATABASE.

The first AFTER trigger nested inside an INSTEAD OF trigger fires even if the **nested triggers** server configuration option is set to 0. However, under this setting, later AFTER triggers do not fire. We recommend that you review your applications for nested triggers to determine whether

the applications comply with your business rules with regard to this behavior when the **nested triggers** server configuration option is set to 0, and then make appropriate modifications.

Deferred Name Resolution

SQL Server allows for Transact-SQL stored procedures, triggers, and batches to refer to tables that do not exist at compile time. This ability is called deferred name resolution.

Permissions

To create a DML trigger requires ALTER permission on the table or view on which the trigger is being created.

To create a DDL trigger with server scope (ON ALL SERVER) or a logon trigger requires CONTROL SERVER permission on the server. To create a DDL trigger with database scope (ON DATABASE) requires ALTER ANY DATABASE DDL TRIGGER permission in the current database.

Examples

A. Using a DML trigger with a reminder message

The following DML trigger prints a message to the client when anyone tries to add or change data in the `Customer` table.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID ('Sales.reminder1', 'TR') IS NOT NULL
   DROP TRIGGER Sales.reminder1;
GO
CREATE TRIGGER reminder1
ON Sales.Customer
AFTER INSERT, UPDATE
AS RAISERROR ('Notify Customer Relations', 16, 10);
GO
```

B. Using a DML trigger with a reminder e-mail message

The following example sends an e-mail message to a specified person (`MaryM`) when the `Customer` table changes.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID ('Sales.reminder2','TR') IS NOT NULL
    DROP TRIGGER Sales.reminder2;
GO
CREATE TRIGGER reminder2
ON Sales.Customer
```

```
AFTER INSERT, UPDATE, DELETE
AS
   EXEC msdb.dbo.sp_send_dbmail
        @profile_name = 'AdventureWorks2012 Administrator',
        @recipients = 'danw@Adventure-Works.com',
        @body = 'Don''t forget to print a report for the sales force.',
        @subject = 'Reminder';
GO
```

C. Using a DML AFTER trigger to enforce a business rule between the PurchaseOrderHeader and Vendor tables

Because CHECK constraints can reference only the columns on which the column-level or table-level constraint is defined, any cross-table constraints (in this case, business rules) must be defined as triggers.

The following example creates a DML trigger. This trigger checks to make sure the credit rating for the vendor is good when an attempt is made to insert a new purchase order into the `PurchaseOrderHeader` table. To obtain the credit rating of the vendor, the `Vendor` table must be referenced. If the credit rating is too low, a message is displayed and the insertion does not execute.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID ('Purchasing.LowCredit','TR') IS NOT NULL
   DROP TRIGGER Purchasing.LowCredit;
GO
-- This trigger prevents a row from being inserted in the
Purchasing.PurchaseOrderHeader table
-- when the credit rating of the specified vendor is set to 5 (below
average).

CREATE TRIGGER Purchasing.LowCredit ON Purchasing.PurchaseOrderHeader
AFTER INSERT
AS
IF EXISTS (SELECT *
           FROM Purchasing.PurchaseOrderHeader p
           JOIN inserted AS i
           ON p.PurchaseOrderID = i.PurchaseOrderID
           JOIN Purchasing.Vendor AS v
           ON v.BusinessEntityID = p.VendorID
```

```
           WHERE v.CreditRating = 5
          )
BEGIN
RAISERROR ('A vendor''s credit rating is too low to accept new
purchase orders.', 16, 1);
ROLLBACK TRANSACTION;
RETURN
END;
GO

-- This statement attempts to insert a row into the PurchaseOrderHeader table
-- for a vendor that has a below average credit rating.
-- The AFTER INSERT trigger is fired and the INSERT transaction is rolled
back.

INSERT INTO Purchasing.PurchaseOrderHeader (RevisionNumber, Status,
EmployeeID,
VendorID, ShipMethodID, OrderDate, ShipDate, SubTotal, TaxAmt, Freight)
VALUES (
2
,3
,261
,1652
,4
,GETDATE()
,GETDATE()
,44594.55
,3567.564
,1114.8638 );
GO
```

D. Using a database-scoped DDL trigger

The following example uses a DDL trigger to prevent any synonym in a database from being dropped.

```
USE AdventureWorks2012;
GO
IF EXISTS (SELECT * FROM sys.triggers
```

```
    WHERE parent_class = 0 AND name = 'safety')
DROP TRIGGER safety
ON DATABASE;
GO
CREATE TRIGGER safety
ON DATABASE
FOR DROP_SYNONYM
AS
   RAISERROR ('You must disable Trigger "safety" to drop synonyms!',10, 1)
   ROLLBACK
GO
DROP TRIGGER safety
ON DATABASE;
GO
```

E. Using a server-scoped DDL trigger

The following example uses a DDL trigger to print a message if any CREATE DATABASE event occurs on the current server instance, and uses the `EVENTDATA` function to retrieve the text of the corresponding Transact-SQL statement.

Note
For more examples that use EVENTDATA in DDL triggers, see Using the eventdata Function.

```
IF EXISTS (SELECT * FROM sys.server_triggers
    WHERE name = 'ddl_trig_database')
DROP TRIGGER ddl_trig_database
ON ALL SERVER;
GO
CREATE TRIGGER ddl_trig_database
ON ALL SERVER
FOR CREATE_DATABASE
AS
    PRINT 'Database Created.'
    SELECT
EVENTDATA().value('(/EVENT_INSTANCE/TSQLCommand/CommandText)[1]','nvarchar(ma
x)')
GO
DROP TRIGGER ddl_trig_database
```

```
ON ALL SERVER;
GO
```

F. Using a logon trigger

The following logon trigger example denies an attempt to log in to SQL Server as a member of the login_test login if there are already three user sessions running under that login.

```
USE master;
GO
CREATE LOGIN login_test WITH PASSWORD = '3KHJ6dhx(0xVYsdf' MUST_CHANGE,
    CHECK_EXPIRATION = ON;
GO
GRANT VIEW SERVER STATE TO login_test;
GO
CREATE TRIGGER connection_limit_trigger
ON ALL SERVER WITH EXECUTE AS 'login_test'
FOR LOGON
AS
BEGIN
IF ORIGINAL_LOGIN()= 'login_test' AND
    (SELECT COUNT(*) FROM sys.dm_exec_sessions
            WHERE is_user_process = 1 AND
                original_login_name = 'login_test') > 3
    ROLLBACK;
END;
```

G. Viewing the events that cause a trigger to fire

The following example queries the `sys.triggers` and `sys.trigger_events` catalog views to determine which Transact-SQL language events cause trigger `safety` to fire. `safety` is created in the previous example.

```
SELECT TE.*
FROM sys.trigger_events AS TE
JOIN sys.triggers AS T
ON T.object_id = TE.object_id
WHERE T.parent_class = 0
AND T.name = 'safety'
GO
```

See Also

ALTER TABLE
ALTER TRIGGER
COLUMNS_UPDATED
CREATE TABLE
DROP TRIGGER
ENABLE TRIGGER
DISABLE TRIGGER
TRIGGER_NESTLEVEL
EVENTDATA
sys.dm_sql_referenced_entities
sys.dm_sql_referencing_entities
sys.sql_expression_dependencies (Transact-SQL)
sp_help
sp_helptrigger
sp_helptext
sp_rename
sp_settriggerorder
UPDATE()
Getting Information About DML Triggers
Getting Information about DDL Triggers
sys.triggers
sys.trigger_events
sys.sql_modules
sys.assembly_modules
sys.server_triggers
sys.server_trigger_events
sys.server_sql_modules
sys.server_assembly_modules

CREATE TYPE

Creates an alias data type or a user-defined type in the current database. The implementation of an alias data type is based on a SQL Server native system type. A user-defined type is implemented through a class of an assembly in the Microsoft .NET Framework common language runtime (CLR). To bind a user-defined type to its implementation, the CLR assembly that contains the implementation of the type must first be registered in SQL Server by using CREATE ASSEMBLY.

Note
The ability to run CLR code is off by default in SQL Server. You can create, modify and drop database objects that reference managed code modules, but these references will not execute in SQL Server unless the clr enabled Option is enabled by using sp_configure.

Transact-SQL Syntax Conventions

Syntax

```
CREATE TYPE [ schema_name. ] type_name
{
   FROM base_type
   [ ( precision [ , scale ] ) ]
   [ NULL | NOT NULL ]
  | EXTERNAL NAME assembly_name [ .class_name ]
  | AS TABLE ( { <column_definition> | <computed_column_definition> }
       [ <table_constraint> ] [ ,...n ] )
} [ ; ]

<column_definition> ::=
column_name <data_type>
   [ COLLATE collation_name ]
   [ NULL | NOT NULL ]
   [
      DEFAULT constant_expression ]
     | [ IDENTITY [ ( seed ,increment ) ]
   ]
   [ ROWGUIDCOL ] [ <column_constraint> [ ...n ] ]

<data type> ::=
[ type_schema_name . ] type_name
   [ ( precision [ , scale ] | max |
            [ { CONTENT | DOCUMENT } ] xml_schema_collection ) ]

<column_constraint> ::=
{    { PRIMARY KEY | UNIQUE }
```

```
      [ CLUSTERED | NONCLUSTERED ]
      [
         WITH ( <index_option> [ ,...n ] )
      ]
  | CHECK ( logical_expression )
}

<computed_column_definition> ::=
column_name AS computed_column_expression
[ PERSISTED [ NOT NULL ] ]
[
   { PRIMARY KEY | UNIQUE }
      [ CLUSTERED | NONCLUSTERED ]
      [
         WITH ( <index_option> [ ,...n ] )
      ]
   | CHECK ( logical_expression )
]

<table_constraint> ::=
{
   { PRIMARY KEY | UNIQUE }
      [ CLUSTERED | NONCLUSTERED ]
                    ( column [ ASC | DESC ] [ ,...n ] )
      [
        WITH ( <index_option> [ ,...n ] )
      ]
   | CHECK ( logical_expression )
}

<index_option> ::=
{
   IGNORE_DUP_KEY = { ON | OFF }
}
```

Arguments

schema_name

Is the name of the schema to which the alias data type or user-defined type belongs.

type_name

Is the name of the alias data type or user-defined type. Type names must comply with the rules for identifiers.

base_type

Is the SQL Server supplied data type on which the alias data type is based. base_type is **sysname**, with no default, and can be one of the following values:

bigint	**binary(*n*)**	**bit**	**char(*n*)**
date	**datetime**	**datetime2**	**datetimeoffset**
decimal	**float**	**image**	**int**
money	**nchar(*n*)**	**ntext**	**numeric**
nvarchar(*n* | max)	**real**	**smalldatetime**	**smallint**
smallmoney	**sql_variant**	**text**	**time**
tinyint	**uniqueidentifier**	**varbinary(*n* | max)**	**varchar(*n* | max)**

base_type can also be any data type synonym that maps to one of these system data types.

precision

For **decimal** or **numeric**, is a non-negative integer that indicates the maximum total number of decimal digits that can be stored, both to the left and to the right of the decimal point. For more information, see decimal and numeric.

scale

For **decimal** or **numeric**, is a non-negative integer that indicates the maximum number of decimal digits that can be stored to the right of the decimal point, and it must be less than or equal to the precision. For more information, see decimal and numeric.

NULL | NOT NULL

Specifies whether the type can hold a null value. If not specified, NULL is the default.

assembly_name

Specifies the SQL Server assembly that references the implementation of the user-defined type in the common language runtime. assembly_name should match an existing assembly in SQL Server in the current database.

Note

EXTERNAL_NAME is not available in a contained database.

[.class_name]

Specifies the class within the assembly that implements the user-defined type. class_name must be a valid identifier and must exist as a class in the assembly with assembly visibility. class_name is case-sensitive, regardless of the database collation, and must exactly match the class name in the corresponding assembly. The class name can be a namespace-qualified name enclosed in square brackets (**[ ]**) if the programming language that is used to write the class uses the concept of namespaces, such as C#. If class_name is not specified, SQL Server assumes it is the same as type_name.

<column_definition>

Defines the columns for a user-defined table type.

<data type>

Defines the data type in a column for a user-defined table type. For more information about data types, see Data Types (Transact-SQL). For more information about tables, see CREATE TABLE (Transact-SQL).

<column_constraint>

Defines the column constraints for a user-defined table type. Supported constraints include PRIMARY KEY, UNIQUE, and CHECK. For more information about tables, see CREATE TABLE (Transact-SQL).

<computed_column_definition>

Defines a computed column expression as a column in a user-defined table type. For more information about tables, see CREATE TABLE (Transact-SQL).

<table_constraint>

Defines a table constraint on a user-defined table type. Supported constraints include PRIMARY KEY, UNIQUE, and CHECK.

<index_option>

Specifies the error response to duplicate key values in a multiple-row insert operation on a unique clustered or unique nonclustered index. For more information about index options, see CREATE INDEX (Transact-SQL).

Remarks

When CREATE TYPE is used to create a CLR user-defined type, the database compatibility must be 90.

The class of the assembly that is referenced in assembly_name, together with its methods, should satisfy all the requirements for implementing a user-defined type in SQL Server. For more information about these requirements, see CLR User-defined Types.

Additional considerations include the following:

- The class can have overloaded methods, but these methods can be called only from within managed code, not from Transact-SQL.

- Any static members must be declared as **const** or **readonly** if assembly_name is SAFE or EXTERNAL_ACCESS.

Within a database, there can be only one user-defined type registered against any specified type that has been uploaded in SQL Server from the CLR. If a user-defined type is created on a CLR type for which a user-defined type already exists in the database, CREATE TYPE fails with an error. This restriction is required to avoid ambiguity during SQL Type resolution if a CLR type can be mapped to more than one user-defined type.

If any mutator method in the type does not return *void*, the CREATE TYPE statement does not execute.

To modify a user-defined type, you must drop the type by using a DROP TYPE statement and then re-create it.

Unlike user-defined types that are created by using **sp_addtype**, the **public** database role is not automatically granted REFERENCES permission on types that are created by using CREATE TYPE. This permission must be granted separately.

In user-defined table types, structured user-defined types that are used in column_name <data type> are part of the database schema scope in which the table type is defined. To access structured user-defined types in a different scope within the database, use two-part names.

In user-defined table types, the primary key on computed columns must be PERSISTED and NOT NULL.

Permissions

Requires CREATE TYPE permission in the current database and ALTER permission on schema_name. If schema_name is not specified, the default name resolution rules for determining the schema for the current user apply. If assembly_name is specified, a user must either own the assembly or have REFERENCES permission on it.

Examples

A. Creating an alias type based on the varchar data type

The following example creates an alias type based on the system-supplied `varchar` data type.

```
CREATE TYPE SSN

FROM varchar(11) NOT NULL ;
```

B. Creating a user-defined type

The following example creates a type `Utf8String` that references class `utf8string` in the assembly `utf8string`. Before creating the type, assembly `utf8string` is registered in the local database.

```
CREATE ASSEMBLY utf8string

FROM '\\ComputerName\utf8string\utf8string.dll' ;

GO

CREATE TYPE Utf8String
```

```
EXTERNAL NAME utf8string.[Microsoft.Samples.SqlServer.utf8string] ;
GO
```

C. Creating a user-defined table type

The following example creates a user-defined table type that has two columns. For more information about how to create and use table-valued parameters, see Table-valued Parameters (Database Engine).

```
/* Create a user-defined table type */
CREATE TYPE LocationTableType AS TABLE
    ( LocationName VARCHAR(50)
    , CostRate INT )
GO
```

See Also

CREATE ASSEMBLY

DROP TYPE (Transact-SQL)

EVENTDATA (Transact-SQL)

CREATE USER

Adds a user to the current database. There are eleven types of users:

Users based on logins in master This is the most common type of user.

- User based on a login based on a Windows user.
- User based on a login based on a Windows group.
- User based on a login using SQL Server authentication.

Users that authenticate at the database Only allowed in a contained database.

- User based on a Windows user that has no login.
- User based on a Windows group that has no login.
- Contained database user with password.

Users based on Windows principals that connect through Windows group logins

- User based on a Windows user that has no login, but can connect to the Database Engine through membership in a Windows group.
- User based on a Windows group that has no login, but can connect to the Database Engine through membership in a different Windows group.

Users that cannot authenticate These users cannot login to SQL Server.

- User without a login. Cannot login but can be granted permissions.
- User based on a certificate. Cannot login but can be granted permissions and can sign modules.

- User based on an asymmetric key. Cannot login but can be granted permissions and can sign modules.

 Transact-SQL Syntax Conventions

Syntax

Users based on logins in master

```
CREATE USER user_name
   [
      { FOR | FROM } LOGIN login_name
   ]
   [ WITH DEFAULT_SCHEMA = schema_name ]
[ ; ]
```

Users that authenticate at the database

```
CREATE USER
   {
    windows_principal [ WITH <options_list> [ ,... ] ]
   | user_name WITH PASSWORD = 'password' [ , <options_list> [ ,... ]
   }
 [ ; ]
```

Users based on Windows principals that connect through Windows group logins

```
CREATE USER
   {
       windows_principal [ { FOR | FROM } LOGIN windows_principal ]
     | user_name { FOR | FROM } LOGIN windows_principal
   }
   [ WITH DEFAULT_SCHEMA = schema_name ]
[ ; ]
```

Users that cannot authenticate

```
CREATE USER user_name
   {
       WITHOUT LOGIN [ WITH DEFAULT_SCHEMA = schema_name ]
     | { FOR | FROM } CERTIFICATE cert_name
```

```
    | { FOR | FROM } ASYMMETRIC KEY asym_key_name
  }
[ ; ]
```

<options_list> ::=

```
    DEFAULT_SCHEMA = schema_name
  | DEFAULT_LANGUAGE = { NONE | lcid | language name | language alias }
  | SID = sid
```

Arguments

user_name

Specifies the name by which the user is identified inside this database. user_name is a **sysname**. It can be up to 128 characters long. When creating a user based on a Windows principal, the Windows principal name becomes the user name unless another user name is specified.

LOGIN login_name

Specifies the login for which the database user is being created. login_name must be a valid login in the server. Can be a login based on a Windows principal (user or group), or a login using SQL Server authentication. When this SQL Server login enters the database, it acquires the name and ID of the database user that is being created. When creating a login mapped from a Windows principal, use the format **[**<domainName>****<loginName>**]**. For examples, see Syntax Summary.

WITH DEFAULT_SCHEMA = schema_name

Specifies the first schema that will be searched by the server when it resolves the names of objects for this database user.

windows_principal'

Specifies the Windows principal for which the database user is being created. The windows_principal can be a Windows user, or a Windows group. The user will be created even if the windows_principal does not have a login. When connecting to SQL Server, if the windows_principal does not have a login, the Windows principal must authenticate at the Database Engine through membership in a Windows group that has a login, or the connection string must specify the contained database as the initial catalog. When creating a user from a Windows principal, use the format **[**<domainName>****<loginName>**]**. For examples, see Syntax Summary.

WITH PASSWORD = 'password'

Can only be used in a contained database. Specifies the password for the user that is being created.

WITHOUT LOGIN

Specifies that the user should not be mapped to an existing login.

CERTIFICATE cert_name

Specifies the certificate for which the database user is being created.

ASYMMETRIC KEY asym_key_name

Specifies the asymmetric key for which the database user is being created.

DEFAULT_LANGUAGE = { NONE | <lcid> | <language name> | <language alias> }

Specifies the default language for the new user. If a default language is specified for the user and the default language of the database is later changed, the users default language remains as specified. If no default language is specified, the default language for the user will be the default language of the database. If the default language for the user is not specified and the default language of the database is later changed, the default language of the user will change to the new default language for the database.

Important

DEFAULT_LANGUAGE is used only for a contained database user.

SID = sid

Applies only to users with passwords (SQL Server authentication) in a contained database. Specifies the SID of the new database user. If this option is not selected, SQL Server automatically assigns a SID. Use the SID parameter to create users in multiple databases that have the same identity (SID). This is useful when creating users in multiple databases to prepare for AlwaysOn failover. To determine the SID of a user, query sys.database_principals.

Remarks

If FOR LOGIN is omitted, the new database user will be mapped to the SQL Server login with the same name.

The default schema will be the first schema that will be searched by the server when it resolves the names of objects for this database user. Unless otherwise specified, the default schema will be the owner of objects created by this database user.

If the user has a default schema, that default schema will used. If the user does not have a default schema, but the user is a member of a group that has a default schema, the default schema of the group will be used. If the user does not have a default schema, and is a member of more than one group that has a default schema, the schema of the Windows group with the lowest **principle_id** will be used. (It is not possible to explicitly select one of the available default schemas as the preferred schema.) If no default schema can be determined for a user, the **dbo** schema will be used.

DEFAULT_SCHEMA can be set before the schema that it points to is created.

DEFAULT_SCHEMA cannot be specified when you are creating a user mapped to a certificate, or an asymmetric key.

The value of DEFAULT_SCHEMA is ignored if the user is a member of the sysadmin fixed server role. All members of the sysadmin fixed server role have a default schema of `dbo`.

The WITHOUT LOGIN clause creates a user that is not mapped to a SQL Server login. It can connect to other databases as guest. Permissions can be assigned to this user without login and when the security context is changed to a user without login, the original users receives the permissions of the user without login. See example D. Creating and using a user without a login.

Only users that are mapped to Windows principals can contain the backslash character (****).

CREATE USER cannot be used to create a guest user because the guest user already exists inside every database. You can enable the guest user by granting it CONNECT permission, as shown:

```
GRANT CONNECT TO guest;
GO
```

Information about database users is visible in the sys.database_principals catalog view.

Syntax Summary

Users based on logins in master

The following list shows possible syntax for users based on logins. The default schema options are not listed.

- `CREATE USER [Domain1\WindowsUserBarry]`
- `CREATE USER [Domain1\WindowsUserBarry] FOR LOGIN Domain1\WindowsUserBarry`
- `CREATE USER [Domain1\WindowsUserBarry] FROM LOGIN Domain1\WindowsUserBarry`
- `CREATE USER [Domain1\WindowsGroupManagers]`
- `CREATE USER [Domain1\WindowsGroupManagers] FOR LOGIN [Domain1\WindowsGroupManagers]`
- `CREATE USER [Domain1\WindowsGroupManagers] FROM LOGIN [Domain1\WindowsGroupManagers]`
- `CREATE USER SQLAUTHLOGIN`
- `CREATE USER SQLAUTHLOGIN FOR LOGIN SQLAUTHLOGIN`
- `CREATE USER SQLAUTHLOGIN FROM LOGIN SQLAUTHLOGIN`

Users that authenticate at the database

The following list shows possible syntax for users that can only be used in a contained database. The users created will not be related to any logins in the **master** database. The default schema and language options are not listed.

noteDXDOC112778PADS Security Note

This syntax grants users access to the database and also grants new access to the Database Engine.

- `CREATE USER [Domain1\WindowsUserBarry]`
- `CREATE USER [Domain1\WindowsGroupManagers]`
- `CREATE USER Barry WITH PASSWORD = 'sdjklalie8rew8337!$d'`

Users based on Windows principals without logins in master

The following list shows possible syntax for users that have access to the Database Engine through a Windows group but do not have a login in **master**. This syntax can be used in all types of databases. The default schema and language options are not listed.

This syntax is similar to users based on logins in master, but this category of user does not have a login in master. The user must have access to the Database Engine through a Windows group login.

This syntax is similar to contained database users based on Windows principals, but this category of user does not get new access to the Database Engine.

- `CREATE USER [Domain1\WindowsUserBarry]`
- `CREATE USER [Domain1\WindowsUserBarry] FOR LOGIN Domain1\WindowsUserBarry`
- `CREATE USER [Domain1\WindowsUserBarry] FROM LOGIN Domain1\WindowsUserBarry`
- `CREATE USER [Domain1\WindowsGroupManagers]`
- `CREATE USER [Domain1\WindowsGroupManagers] FOR LOGIN [Domain1\WindowsGroupManagers]`
- `CREATE USER [Domain1\WindowsGroupManagers] FROM LOGIN [Domain1\WindowsGroupManagers]`

Users that cannot authenticate

The following list shows possible syntax for users that cannot login to SQL Server.

- `CREATE USER RIGHTSHOLDER WITHOUT LOGIN`
- `CREATE USER CERTUSER FOR CERTIFICATE SpecialCert`
- `CREATE USER CERTUSER FROM CERTIFICATE SpecialCert`
- `CREATE USER KEYUSER FOR ASYMMETRIC KEY SecureKey`
- `CREATE USER KEYUSER FROM ASYMMETRIC KEY SecureKey`

Security

Creating a user grants access to a database but does not automatically grant any access to the objects in a database. After creating a user, common actions are to add users to database roles which have permission to access database objects, or grant object permissions to the user.

Special Considerations for Contained Databases

When connecting to a contained database, if the user does not have a login in the **master** database, the connection string must include the contained database name as the initial catalog. The initial catalog parameter is always required for a contained database user with password.

In a contained database, creating users helps separate the database from the instance of the Database Engine so that the database can easily be moved to another instance of SQL Server. For more information, see Understanding Contained Databases. To change a database user from a user based on a SQL Server authentication login to a contained database user with password, see sp_migrate_user_to_contained (Transact-SQL).

In a contained database, users do not have to have logins in the **master** database. Database Engine administrators should understand that access to a contained database can be granted at the database level, instead of the Database Engine level. For more information, see Threats Against Contained Databases.

Permissions

Requires ALTER ANY USER permission on the database.

Examples

A. Creating a database user based on a SQL Server login

The following example first creates a SQL Server login named `AbolrousHazem`, and then creates a corresponding database user `AbolrousHazem` in `AdventureWorks2012`.

```
CREATE LOGIN AbolrousHazem
    WITH PASSWORD = '340$Uuxwp7Mcxo7Khy';
USE AdventureWorks2012;
GO
CREATE USER AbolrousHazem FOR LOGIN AbolrousHazem;
GO
```

B. Creating a database user with a default schema

The following example first creates a server login named `WanidaBenshoof` with a password, and then creates a corresponding database user `Wanida`, with the default schema `Marketing`.

```
CREATE LOGIN WanidaBenshoof
    WITH PASSWORD = '8fdKJl3$nlNv3049jsKK';
USE AdventureWorks2012;
CREATE USER Wanida FOR LOGIN WanidaBenshoof
    WITH DEFAULT_SCHEMA = Marketing;
GO
```

C. Creating a database user from a certificate

The following example creates a database user `JinghaoLiu` from certificate `CarnationProduction50`.

```
USE AdventureWorks2012;
CREATE CERTIFICATE CarnationProduction50
    WITH SUBJECT = 'Carnation Production Facility Supervisors',
    EXPIRY_DATE = '11/11/2011';
GO
CREATE USER JinghaoLiu FOR CERTIFICATE CarnationProduction50;
```

```
GO
```

D. Creating and using a user without a login

The following example creates a database user `CustomApp` that does not map to a SQL Server login. The example then grants a user `adventure-works\tengiz0` permission to impersonate the `CustomApp` user.

```
USE AdventureWorks2012 ;

CREATE USER CustomApp WITHOUT LOGIN ;

GRANT IMPERSONATE ON USER::CustomApp TO [adventure-works\tengiz0] ;

GO
```

To use the `CustomApp` credentials, the user `adventure-works\tengiz0` executes the following statement.

```
EXECUTE AS USER = 'CustomApp' ;

GO
```

To revert back to the `adventure-works\tengiz0` credentials, the user executes the following statement.

```
REVERT ;

GO
```

E. Creating a contained database user with password

The following example creates a contained database user with password. This example can only be executed in a contained database.

```
USE AdventureWorks2012 ;

GO

CREATE USER Carlo

WITH PASSWORD='RN92piTCh%$!~3K9844 Bl*'

    , DEFAULT_LANGUAGE=[Brazilian]

    , DEFAULT_SCHEMA=[dbo]

GO
```

F. Creating a contained database user for a domain login

The following example creates a contained database user for a login named Fritz in a domain named Contoso. This example can only be executed in a contained database.

```
USE AdventureWorks2012 ;

GO

CREATE USER [Contoso\Fritz] ;

GO
```

G. Creating a contained database user with a specific SID

The following example creates a SQL Server authenticated contained database user named CarmenW. This example can only be executed in a contained database.

```
USE AdventureWorks2012 ;
GO
CREATE USER CarmenW WITH PASSWORD = 'a8ea v*(Rd##+'
, SID = 0x0105000000000009030000063FF0451A9E7664BA705B10E37DDC4B7;
```

See Also

Creating a Database User
sys.database_principals (Transact-SQL)
ALTER USER (Transact-SQL)
DROP USER (Transact-SQL)
CREATE LOGIN (Transact-SQL)
eventdata (Transact-SQL)
Understanding Contained Databases

CREATE VIEW

Creates a virtual table whose contents (columns and rows) are defined by a query. Use this statement to create a view of the data in one or more tables in the database. For example, a view can be used for the following purposes:

- To focus, simplify, and customize the perception each user has of the database.
- As a security mechanism by allowing users to access data through the view, without granting the users permissions to directly access the underlying base tables.
- To provide a backward compatible interface to emulate a table whose schema has changed.

Transact-SQL Syntax Conventions

Syntax

```
CREATE VIEW [ schema_name . ] view_name [ (column [ ,...n ] ) ]
[ WITH <view_attribute> [ ,...n ] ]
AS select_statement
[ WITH CHECK OPTION ] [ ; ]

<view_attribute> ::=
{
```

[ENCRYPTION]
[SCHEMABINDING]
[VIEW_METADATA] }

Arguments

schema_name

Is the name of the schema to which the view belongs.

view_name

Is the name of the view. View names must follow the rules for identifiers. Specifying the view owner name is optional.

column

Is the name to be used for a column in a view. A column name is required only when a column is derived from an arithmetic expression, a function, or a constant; when two or more columns may otherwise have the same name, typically because of a join; or when a column in a view is specified a name different from that of the column from which it is derived. Column names can also be assigned in the SELECT statement.

If column is not specified, the view columns acquire the same names as the columns in the SELECT statement.

Note

In the columns for the view, the permissions for a column name apply across a CREATE VIEW or ALTER VIEW statement, regardless of the source of the underlying data. For example, if permissions are granted on the **SalesOrderID** column in a CREATE VIEW statement, an ALTER VIEW statement can name the **SalesOrderID** column with a different column name, such as **OrderRef**, and still have the permissions associated with the view using **SalesOrderID**.

AS

Specifies the actions the view is to perform.

select_statement

Is the SELECT statement that defines the view. The statement can use more than one table and other views. Appropriate permissions are required to select from the objects referenced in the SELECT clause of the view that is created.

A view does not have to be a simple subset of the rows and columns of one particular table. A view can be created that uses more than one table or other views with a SELECT clause of any complexity.

In an indexed view definition, the SELECT statement must be a single table statement or a multitable JOIN with optional aggregation.

The SELECT clauses in a view definition cannot include the following:

- An ORDER BY clause, unless there is also a TOP clause in the select list of the SELECT statement

Important

The ORDER BY clause is used only to determine the rows that are returned by the TOP or OFFSET clause in the view definition. The ORDER BY clause does not guarantee ordered results when the view is queried, unless ORDER BY is also specified in the query itself.

- The INTO keyword
- The OPTION clause
- A reference to a temporary table or a table variable.

Because select_statement uses the SELECT statement, it is valid to use <join_hint> and <table_hint> hints as specified in the FROM clause. For more information, see View Resolution and SELECT (Transact-SQL).

Functions and multiple SELECT statements separated by UNION or UNION ALL can be used in select_statement.

CHECK OPTION

Forces all data modification statements executed against the view to follow the criteria set within select_statement. When a row is modified through a view, the WITH CHECK OPTION makes sure the data remains visible through the view after the modification is committed.

Note

Any updates performed directly to a view's underlying tables are not verified against the view, even if CHECK OPTION is specified.

ENCRYPTION

Encrypts the entries in sys.syscomments that contain the text of the CREATE VIEW statement. Using WITH ENCRYPTION prevents the view from being published as part of SQL Server replication.

SCHEMABINDING

Binds the view to the schema of the underlying table or tables. When SCHEMABINDING is specified, the base table or tables cannot be modified in a way that would affect the view definition. The view definition itself must first be modified or dropped to remove dependencies on the table that is to be modified. When you use SCHEMABINDING, the select_statement must include the two-part names (schema**.**object) of tables, views, or user-defined functions that are referenced. All referenced objects must be in the same database.

Views or tables that participate in a view created with the SCHEMABINDING clause cannot be dropped unless that view is dropped or changed so that it no longer has schema binding. Otherwise, the Database Engine raises an error. Also, executing ALTER TABLE statements on tables that participate in views that have schema binding fail when these statements affect the view definition.

VIEW_METADATA

Specifies that the instance of SQL Server will return to the DB-Library, ODBC, and OLE DB APIs the metadata information about the view, instead of the base table or tables, when

browse-mode metadata is being requested for a query that references the view. Browse-mode metadata is additional metadata that the instance of SQL Server returns to these client-side APIs. This metadata enables the client-side APIs to implement updatable client-side cursors. Browse-mode metadata includes information about the base table that the columns in the result set belong to.

For views created with VIEW_METADATA, the browse-mode metadata returns the view name and not the base table names when it describes columns from the view in the result set.

When a view is created by using WITH VIEW_METADATA, all its columns, except a **timestamp** column, are updatable if the view has INSTEAD OF INSERT or INSTEAD OF UPDATE triggers. For more information about updatable views, see Remarks.

Remarks

A view can be created only in the current database. The CREATE VIEW must be the first statement in a query batch. A view can have a maximum of 1,024 columns.

When querying through a view, the Database Engine checks to make sure that all the database objects referenced anywhere in the statement exist and that they are valid in the context of the statement, and that data modification statements do not violate any data integrity rules. A check that fails returns an error message. A successful check translates the action into an action against the underlying table or tables.

If a view depends on a table or view that was dropped, the Database Engine produces an error message when anyone tries to use the view. If a new table or view is created and the table structure does not change from the previous base table to replace the one dropped, the view again becomes usable. If the new table or view structure changes, the view must be dropped and re-created.

If a view is not created with the SCHEMABINDING clause, sp_refreshview should be run when changes are made to the objects underlying the view that affect the definition of the view. Otherwise, the view might produce unexpected results when it is queried.

When a view is created, information about the view is stored in the following catalog views: sys.views, sys.columns, and sys.sql_expression_dependencies. The text of the CREATE VIEW statement is stored in the sys.sql_modules catalog view.

A query that uses an index on a view defined with **numeric** or **float** expressions may have a result that is different from a similar query that does not use the index on the view. This difference may be caused by rounding errors during INSERT, DELETE, or UPDATE actions on underlying tables.

The Database Engine saves the settings of SET QUOTED_IDENTIFIER and SET ANSI_NULLS when a view is created. These original settings are used to parse the view when the view is used. Therefore, any client-session settings for SET QUOTED_IDENTIFIER and SET ANSI_NULLS do not affect the view definition when the view is accessed.

Updatable Views

You can modify the data of an underlying base table through a view, as long as the following conditions are true:

- Any modifications, including UPDATE, INSERT, and DELETE statements, must reference columns from only one base table.
- The columns being modified in the view must directly reference the underlying data in the table columns. The columns cannot be derived in any other way, such as through the following:
  - An aggregate function: AVG, COUNT, SUM, MIN, MAX, GROUPING, STDEV, STDEVP, VAR, and VARP.
  - A computation. The column cannot be computed from an expression that uses other columns. Columns that are formed by using the set operators UNION, UNION ALL, CROSSJOIN, EXCEPT, and INTERSECT amount to a computation and are also not updatable.
- The columns being modified are not affected by GROUP BY, HAVING, or DISTINCT clauses.
- TOP is not used anywhere in the select_statement of the view together with the WITH CHECK OPTION clause.

The previous restrictions apply to any subqueries in the FROM clause of the view, just as they apply to the view itself. Generally, the Database Engine must be able to unambiguously trace modifications from the view definition to one base table. For more information, see Modifying Data Through a View.

If the previous restrictions prevent you from modifying data directly through a view, consider the following options:

- **INSTEAD OF Triggers**

  INSTEAD OF triggers can be created on a view to make a view updatable. The INSTEAD OF trigger is executed instead of the data modification statement on which the trigger is defined. This trigger lets the user specify the set of actions that must happen to process the data modification statement. Therefore, if an INSTEAD OF trigger exists for a view on a specific data modification statement (INSERT, UPDATE, or DELETE), the corresponding view is updatable through that statement. For more information about INSTEAD OF triggers, see DML Triggers.

- **Partitioned Views**

  If the view is a partitioned view, the view is updatable, subject to certain restrictions. When it is needed, the Database Engine distinguishes local partitioned views as the views in which all participating tables and the view are on the same instance of SQL Server, and distributed partitioned views as the views in which at least one of the tables in the view resides on a different or remote server.

Partitioned Views

A partitioned view is a view defined by a UNION ALL of member tables structured in the same way, but stored separately as multiple tables in either the same instance of SQL Server or in a group of autonomous instances of SQL Server servers, called federated database servers.

Note
The preferred method for partitioning data local to one server is through partitioned tables. For more information, see Partitioned Tables and Indexes.

In designing a partitioning scheme, it must be clear what data belongs to each partition. For example, the data for the `Customers` table is distributed in three member tables in three server locations: `Customers_33` on `Server1`, `Customers_66` on `Server2`, and `Customers_99` on `Server3`.

A partitioned view on `Server1` is defined in the following way:

```
--Partitioned view as defined on Server1
CREATE VIEW Customers
AS
--Select from local member table.
SELECT *
FROM CompanyData.dbo.Customers_33
UNION ALL
--Select from member table on Server2.
SELECT *
FROM Server2.CompanyData.dbo.Customers_66
UNION ALL
--Select from mmeber table on Server3.
SELECT *
FROM Server3.CompanyData.dbo.Customers_99;
```

Generally, a view is said to be a partitioned view if it is of the following form:

```
SELECT <select_list1>
FROM T1
UNION ALL
SELECT <select_list2>
FROM T2
UNION ALL
...
SELECT <select_listn>
FROM Tn;
```

Conditions for Creating Partitioned Views

1. The select `list`
   - All columns in the member tables should be selected in the column list of the view definition.
   - The columns in the same ordinal position of each `select list` should be of the same type, including collations. It is not sufficient for the columns to be implicitly convertible types, as is generally the case for UNION.

     Also, at least one column (for example `<col>`) must appear in all the select lists in the same ordinal position. This `<col>` should be defined in a way that the member tables `T1, ..., Tn` have CHECK constraints `C1, ..., Cn` defined on `<col>`, respectively.

     Constraint `C1` defined on table `T1` must be of the following form:

     ```
     C1 ::= < simple_interval > [ OR < simple_interval > OR ...]

     < simple_interval > :: =

     < col > { < | > | <= | >= | = < value >}

     | < col > BETWEEN < value1 > AND < value2 >

     | < col > IN ( value_list )

     | < col > { > | >= } < value1 > AND

     < col > { < | <= } < value2 >
     ```
   - The constraints should be in such a way that any specified value of `<col>` can satisfy, at most, one of the constraints `C1, ..., Cn` so that the constraints should form a set of disjointed or nonoverlapping intervals. The column `<col>` on which the disjointed constraints are defined is called the partitioning column. Note that the partitioning column may have different names in the underlying tables. The constraints should be in an enabled and trusted state for them to meet the previously mentioned conditions of the partitioning column. If the constraints are disabled, re-enable constraint checking by using the CHECK CONSTRAINT constraint_name option of ALTER TABLE, and using the WITH CHECK option to validate them.

     The following examples show valid sets of constraints:

     ```
     { [col < 10], [col between 11 and 20] , [col > 20] }

     { [col between 11 and 20], [col between 21 and 30], [col between 31 and
     100] }
     ```
   - The same column cannot be used multiple times in the select list.
2. Partitioning column
   - The partitioning column is a part of the PRIMARY KEY of the table.
   - It cannot be a computed, identity, default, or **timestamp** column.
   - If there is more than one constraint on the same column in a member table, the Database Engine ignores all the constraints and does not consider them when

determining whether the view is a partitioned view. To meet the conditions of the partitioned view, there should be only one partitioning constraint on the partitioning column.

- There are no restrictions on the updatability of the partitioning column.

3. Member tables, or underlying tables `T1, ..., Tn`
   - The tables can be either local tables or tables from other computers that are running SQL Server that are referenced either through a four-part name or an OPENDATASOURCE- or OPENROWSET-based name. The OPENDATASOURCE and OPENROWSET syntax can specify a table name, but not a pass-through query. For more information, see OPENDATASOURCE (Transact-SQL) and OPENROWSET (Transact-SQL).

     If one or more of the member tables are remote, the view is called distributed partitioned view, and additional conditions apply. They are described later in this section.
   - The same table cannot appear two times in the set of tables that are being combined with the UNION ALL statement.
   - The member tables cannot have indexes created on computed columns in the table.
   - The member tables should have all PRIMARY KEY constraints on the same number of columns.
   - All member tables in the view should have the same ANSI padding setting. This can be set by using either the **user options** option in **sp_configure** or the SET statement.

Conditions for Modifying Data in Partitioned Views

The following restrictions apply to statements that modify data in partitioned views:

- The INSERT statement must supply values for all the columns in the view, even if the underlying member tables have a DEFAULT constraint for those columns or if they allow for null values. For those member table columns that have DEFAULT definitions, the statements cannot explicitly use the keyword DEFAULT.
- The value being inserted into the partitioning column should satisfy at least one of the underlying constraints; otherwise, the insert action will fail with a constraint violation.
- UPDATE statements cannot specify the DEFAULT keyword as a value in the SET clause, even if the column has a DEFAULT value defined in the corresponding member table.
- Columns in the view that are an identity column in one or more of the member tables cannot be modified by using an INSERT or UPDATE statement.
- If one of the member tables contains a **timestamp** column, the data cannot be modified by using an INSERT or UPDATE statement.
- If one of the member tables contains a trigger or an ON UPDATE CASCADE/SET NULL/SET DEFAULT or ON DELETE CASCADE/SET NULL/SET DEFAULT constraint, the view cannot be modified.
- INSERT, UPDATE, and DELETE actions against a partitioned view are not allowed if there is a self-join with the same view or with any of the member tables in the statement.

- Bulk importing data into a partitioned view is unsupported by **bcp** or the BULK INSERT and INSERT ... SELECT * FROM OPENROWSET(BULK...) statements. However, you can insert multiple rows into a partitioned view by using the INSERT statement.

  Note
  To update a partitioned view, the user must have INSERT, UPDATE, and DELETE permissions on the member tables.

Additional Conditions for Distributed Partitioned Views

For distributed partitioned views (when one or more member tables are remote), the following additional conditions apply:

- A distributed transaction will be started to guarantee atomicity across all nodes affected by the update.
- The XACT_ABORT SET option should be set to ON for INSERT, UPDATE, or DELETE statements to work.
- Any columns in remote tables of type **smallmoney** that are referenced in a partitioned view are mapped as **money**. Therefore, the corresponding columns (in the same ordinal position in the select list) in the local tables must also be of type **money**.
- Under database compatibility level 110, any columns in remote tables of type **smalldatetime** that are referenced in a partitioned view are mapped as **smalldatetime**. Corresponding columns (in the same ordinal position in the select list) in the local tables must be **smalldatetime**. This is a change in behavior from earlier versions of SQL Server in which any columns in remote tables of type **smalldatetime** that are referenced in a partitioned view are mapped as **datetime** and corresponding columns in local tables must be of type **datetime**. For more information, see ALTER DATABASE Compatibility Level (Transact-SQL).
- Any linked server in the partitioned view cannot be a loopback linked server. This is a linked server that points to the same instance of SQL Server.

The setting of the SET ROWCOUNT option is ignored for INSERT, UPDATE, and DELETE actions that involve updatable partitioned views and remote tables.

When the member tables and partitioned view definition are in place, the SQL Server query optimizer builds intelligent plans that use queries efficiently to access data from member tables. With the CHECK constraint definitions, the query processor maps the distribution of key values across the member tables. When a user issues a query, the query processor compares the map to the values specified in the WHERE clause, and builds an execution plan with a minimal amount of data transfer between member servers. Therefore, although some member tables may be located in remote servers, the instance of SQL Server resolves distributed queries so that the amount of distributed data that has to be transferred is minimal.

Considerations for Replication

To create partitioned views on member tables that are involved in replication, the following considerations apply:

- If the underlying tables are involved in merge replication or transactional replication with updating subscriptions, the **uniqueidentifier** column should also be included in the select list.

  Any INSERT actions into the partitioned view must provide a NEWID() value for the **uniqueidentifier** column. Any UPDATE actions against the **uniqueidentifier** column must supply NEWID() as the value because the DEFAULT keyword cannot be used.
- The replication of updates made by using the view is the same as when tables are replicated in two different databases: the tables are served by different replication agents and the order of the updates is not guaranteed.

Permissions

Requires CREATE VIEW permission in the database and ALTER permission on the schema in which the view is being created.

Examples

A. Using a simple CREATE VIEW

The following example creates a view by using a simple `SELECT` statement. A simple view is helpful when a combination of columns is queried frequently. The data from this view comes from the `HumanResources.Employee` and `Person.Person` tables of the AdventureWorks2012 database. The data provides name and hire date information for the employees of Adventure Works Cycles. The view could be created for the person in charge of tracking work anniversaries but without giving this person access to all the data in these tables.

```
USE AdventureWorks2012 ;
GO
IF OBJECT_ID ('hiredate_view', 'V') IS NOT NULL
DROP VIEW hiredate_view ;
GO
CREATE VIEW hiredate_view
AS
SELECT p.FirstName, p.LastName, e.BusinessEntityID, e.HireDate
FROM HumanResources.Employee e
JOIN Person.Person AS p ON e.BusinessEntityID = p.BusinessEntityID ;
GO
```

B. Using WITH ENCRYPTION

The following example uses the `WITH ENCRYPTION` option and shows computed columns, renamed columns, and multiple columns.

```
USE AdventureWorks2012 ;
GO
```

```
IF OBJECT_ID ('Purchasing.PurchaseOrderReject', 'V') IS NOT NULL
    DROP VIEW Purchasing.PurchaseOrderReject ;
GO
CREATE VIEW Purchasing.PurchaseOrderReject
WITH ENCRYPTION
AS
SELECT PurchaseOrderID, ReceivedQty, RejectedQty,
    RejectedQty / ReceivedQty AS RejectRatio, DueDate
FROM Purchasing.PurchaseOrderDetail
WHERE RejectedQty / ReceivedQty > 0
AND DueDate > CONVERT(DATETIME,'20010630',101) ;
GO
```

C. Using WITH CHECK OPTION

The following example shows a view named `SeattleOnly` that references five tables and allows for data modifications to apply only to employees who live in Seattle.

```
USE AdventureWorks2012 ;
GO
IF OBJECT_ID ('dbo.SeattleOnly', 'V') IS NOT NULL
    DROP VIEW dbo.SeattleOnly ;
GO
CREATE VIEW dbo.SeattleOnly
AS
SELECT p.LastName, p.FirstName, e.JobTitle, a.City, sp.StateProvinceCode
FROM HumanResources.Employee e
    INNER JOIN Person.Person p
    ON p.BusinessEntityID = e.BusinessEntityID
    INNER JOIN Person.BusinessEntityAddress bea
    ON bea.BusinessEntityID = e.BusinessEntityID
    INNER JOIN Person.Address a
    ON a.AddressID = bea.AddressID
    INNER JOIN Person.StateProvince sp
    ON sp.StateProvinceID = a.StateProvinceID
WHERE a.City = 'Seattle'
WITH CHECK OPTION ;
GO
```

D. Using built-in functions within a view

The following example shows a view definition that includes a built-in function. When you use functions, you must specify a column name for the derived column.

```
USE AdventureWorks2012 ;
GO
IF OBJECT_ID ('Sales.SalesPersonPerform', 'V') IS NOT NULL
    DROP VIEW Sales.SalesPersonPerform ;
GO
CREATE VIEW Sales.SalesPersonPerform
AS
SELECT TOP (100) SalesPersonID, SUM(TotalDue) AS TotalSales
FROM Sales.SalesOrderHeader
WHERE OrderDate > CONVERT(DATETIME,'20001231',101)
GROUP BY SalesPersonID;
GO
```

E. Using partitioned data

The following example uses tables named `SUPPLY1`, `SUPPLY2`, `SUPPLY3`, and `SUPPLY4`. These tables correspond to the supplier tables from four offices, located in different countries/regions.

```
--Create the tables and insert the values.
CREATE TABLE dbo.SUPPLY1 (
supplyID INT PRIMARY KEY CHECK (supplyID BETWEEN 1 and 150),
supplier CHAR(50)
);
CREATE TABLE dbo.SUPPLY2 (
supplyID INT PRIMARY KEY CHECK (supplyID BETWEEN 151 and 300),
supplier CHAR(50)
);
CREATE TABLE dbo.SUPPLY3 (
supplyID INT PRIMARY KEY CHECK (supplyID BETWEEN 301 and 450),
supplier CHAR(50)
);
CREATE TABLE dbo.SUPPLY4 (
supplyID INT PRIMARY KEY CHECK (supplyID BETWEEN 451 and 600),
supplier CHAR(50)
);
```

```
GO
INSERT dbo.SUPPLY1 VALUES ('1', 'CaliforniaCorp'), ('5', 'BraziliaLtd');
INSERT dbo.SUPPLY2 VALUES ('231', 'FarEast'), ('280', 'NZ');
INSERT dbo.SUPPLY3 VALUES ('321', 'EuroGroup'), ('442', 'UKArchip');
INSERT dbo.SUPPLY4 VALUES ('475', 'India'), ('521', 'Afrique');
GO
--Create the view that combines all supplier tables.
CREATE VIEW dbo.all_supplier_view
WITH SCHEMABINDING
AS
SELECT supplyID, supplier
FROM dbo.SUPPLY1
UNION ALL
SELECT supplyID, supplier
FROM dbo.SUPPLY2
UNION ALL
SELECT supplyID, supplier
FROM dbo.SUPPLY3
UNION ALL
SELECT supplyID, supplier
FROM dbo.SUPPLY4;
```

See Also

ALTER TABLE
ALTER VIEW
DELETE (Transact-SQL)
DROP VIEW
INSERT (Transact-SQL)
Create a Stored Procedure
sys.dm_sql_referenced_entities
sys.dm_sql_referencing_entities
sp_help (Transact-SQL)
sp_helptext (Transact-SQL)
sp_refreshview
sp_rename (Transact-SQL)

sys.views (Transact-SQL)
UPDATE (Transact-SQL)
EVENTDATA (Transact-SQL)

CREATE WORKLOAD GROUP

Creates a Resource Governor workload group and associates the workload group with a Resource Governor resource pool. Resource Governor is not available in every edition of Microsoft SQL Server. For a list of features that are supported by the editions of SQL Server, see Features Supported by the Editions of SQL Server 2012.

 Transact-SQL Syntax Conventions.

Syntax

```
CREATE WORKLOAD GROUP group_name
[ WITH
      (  [ IMPORTANCE = { LOW | MEDIUM | HIGH } ]
        [ [ , ] REQUEST_MAX_MEMORY_GRANT_PERCENT = value ]
        [ [ , ] REQUEST_MAX_CPU_TIME_SEC = value ]
        [ [ , ] REQUEST_MEMORY_GRANT_TIMEOUT_SEC = value ]
        [ [ , ] MAX_DOP = value ]
        [ [ , ] GROUP_MAX_REQUESTS = value ] )
 ]
[ USING { pool_name | "default" } ]
[ ; ]
```

Arguments

group_name

Is the user-defined name for the workload group. group_name is alphanumeric, can be up to 128 characters, must be unique within an instance of SQL Server, and must comply with the rules for identifiers.

IMPORTANCE = { LOW | MEDIUM | HIGH }

Specifies the relative importance of a request in the workload group. Importance is one of the following, with MEDIUM being the default:

- LOW
- MEDIUM
- HIGH

 **Note**

Internally each importance setting is stored as a number that is used for calculations.

IMPORTANCE is local to the resource pool; workload groups of different importance inside the same resource pool affect each other, but do not affect workload groups in another resource pool.

REQUEST_MAX_MEMORY_GRANT_PERCENT = value

Specifies the maximum amount of memory that a single request can take from the pool. This percentage is relative to the resource pool size specified by MAX_MEMORY_PERCENT.

Note

The amount specified only refers to query execution grant memory.

value must be 0 or a positive integer. The allowed range for value is from 0 through 100. The default setting for value is 25.

Note the following:

- Setting value to 0 prevents queries with SORT and HASH JOIN operations in user-defined workload groups from running.
- We do not recommend setting value greater than 70 because the server may be unable to set aside enough free memory if other concurrent queries are running. This may eventually lead to query time-out error 8645.

Note

- If the query memory requirements exceed the limit that is specified by this parameter, the server does the following:
- For user-defined workload groups, the server tries to reduce the query degree of parallelism until the memory requirement falls under the limit, or until the degree of parallelism equals 1. If the query memory requirement is still greater than the limit, error 8657 occurs.
- For internal and default workload groups, the server permits the query to obtain the required memory.
- Be aware that both cases are subject to time-out error 8645 if the server has insufficient physical memory.

REQUEST_MAX_CPU_TIME_SEC = value

Specifies the maximum amount of CPU time, in seconds, that a request can use. value must be 0 or a positive integer. The default setting for value is 0, which means unlimited.

Note

Resource Governor will not prevent a request from continuing if the maximum time is exceeded. However, an event will be generated. For more information, see CPU Threshold Exceeded Event Class.

REQUEST_MEMORY_GRANT_TIMEOUT_SEC = value

Specifies the maximum time, in seconds, that a query can wait for a memory grant (work

buffer memory) to become available.

Note

A query does not always fail when memory grant time-out is reached. A query will only fail if there are too many concurrent queries running. Otherwise, the query may only get the minimum memory grant, resulting in reduced query performance.

value must be 0 or a positive integer. The default setting for value, 0, uses an internal calculation based on query cost to determine the maximum time.

MAX_DOP = value

Specifies the maximum degree of parallelism (DOP) for parallel requests. value must be 0 or a positive integer. The allowed range for value is from 0 through 64. The default setting for value, 0, uses the global setting. MAX_DOP is handled as follows:

- MAX_DOP as a query hint is effective as long as it does not exceed workload group MAX_DOP.
- MAX_DOP as a query hint always overrides sp_configure 'max degree of parallelism' in SQL Server 2005.
- Workload group MAX_DOP overrides sp_configure 'max degree of parallelism'.
- If the query is marked as serial at compile time, it cannot be changed back to parallel at run time regardless of the workload group or sp_configure setting.
- After DOP is configured, it can only be lowered on grant memory pressure. Workload group reconfiguration is not visible while waiting in the grant memory queue.

GROUP_MAX_REQUESTS = value

Specifies the maximum number of simultaneous requests that are allowed to execute in the workload group. value must be a 0 or a positive integer. The default setting for value, 0, allows unlimited requests. When the maximum concurrent requests are reached, a user in that group can log in, but is placed in a wait state until concurrent requests are dropped below the value specified.

USING { pool_name | "default" }

Associates the workload group with the user-defined resource pool identified by pool_name. This in effect puts the workload group in the resource pool. If pool_name is not provided, or if the USING argument is not used, the workload group is put in the predefined Resource Governor default pool.

"default" is a reserved word and when used with USING, must be enclosed by quotation marks ("") or brackets ([]).

Note

Predefined workload groups and resource pools all use lower case names, such as "default". This should be taken into account for servers that use case-sensitive collation. Servers with case-insensitive collation, such as SQL_Latin1_General_CP1_CI_AS, will treat "default" and "Default" as the same.

Remarks

REQUEST_MEMORY_GRANT_PERCENT: Index creation is allowed to use more workspace memory than what is initially granted for improved performance. This special handling is supported by Resource Governor in SQL Server 2012. However, the initial grant and any additional memory grant are limited by resource pool and workload group settings.

Index Creation on a Partitioned Table

The memory consumed by index creation on non-aligned partitioned table is proportional to the number of partitions involved. If the total required memory exceeds the per-query limit (REQUEST_MAX_MEMORY_GRANT_PERCENT) imposed by the Resource Governor workload group setting, this index creation may fail to execute. Because the "default" workload group allows a query to exceed the per-query limit with the minimum required memory, the user may be able to run the same index creation in "default" workload group, if the "default" resource pool has enough total memory configured to run such query.

Permissions

Requires CONTROL SERVER permission.

Examples

The following example shows how to create a workload group named `newReports`. It uses the Resource Governor default settings and is in the Resource Governor default pool. The example specifies the `default` pool, but this is not required.

```
CREATE WORKLOAD GROUP newReports
    USING "default" ;
GO
```

See Also

ALTER WORKLOAD GROUP (Transact-SQL)

DROP WORKLOAD GROUP (Transact-SQL)

CREATE RESOURCE POOL (Transact-SQL)

ALTER RESOURCE POOL (Transact-SQL)

DROP RESOURCE POOL (Transact-SQL)

ALTER RESOURCE GOVERNOR (Transact-SQL)

CREATE XML INDEX

Creates an XML index on a specified table. An index can be created before there is data in the table. XML indexes can be created on tables in another database by specifying a qualified database name.

Note

To create a relational index, see CREATE INDEX (Transact-SQL). For information about how to create a spatial index, see CREATE SPATIAL INDEX (Transact-SQL).

 Transact-SQL Syntax Conventions

Syntax

Create XML Index

```
CREATE [ PRIMARY ] XML INDEX index_name
   ON <object> ( xml_column_name )
   [ USING XML INDEX xml_index_name
      [ FOR { VALUE | PATH | PROPERTY } ] ]
   [ WITH ( <xml_index_option> [ ,...n ] ) ]
[ ; ]
```

<object> ::=

```
{
   [ database_name. [ schema_name ] . | schema_name. ]
   table_name
}
```

<xml_index_option> ::=

```
{
   PAD_INDEX  = { ON | OFF }
  | FILLFACTOR = fillfactor
  | SORT_IN_TEMPDB = { ON | OFF }
  | IGNORE_DUP_KEY = OFF
  | DROP_EXISTING = { ON | OFF }
  | ONLINE = OFF
  | ALLOW_ROW_LOCKS = { ON | OFF }
  | ALLOW_PAGE_LOCKS = { ON | OFF }
  | MAXDOP = max_degree_of_parallelism
}
```

Arguments

[PRIMARY] XML

Creates an XML index on the specified **xml** column. When PRIMARY is specified, a clustered

index is created with the clustered key formed from the clustering key of the user table and an XML node identifier. Each table can have up to 249 XML indexes. Note the following when you create an XML index:

- A clustered index must exist on the primary key of the user table.
- The clustering key of the user table is limited to 15 columns.
- Each **xml** column in a table can have one primary XML index and multiple secondary XML indexes.
- A primary XML index on an **xml** column must exist before a secondary XML index can be created on the column.
- An XML index can only be created on a single **xml** column. You cannot create an XML index on a non-**xml** column, nor can you create a relational index on an **xml** column.
- You cannot create an XML index, either primary or secondary, on an **xml** column in a view, on a table-valued variable with **xml** columns, or **xml** type variables.
- You cannot create a primary XML index on a computed **xml** column.
- The SET option settings must be the same as those required for indexed views and computed column indexes. Specifically, the option ARITHABORT must be set to ON when an XML index is created and when inserting, deleting, or updating values in the **xml** column.

For more information, see Indexes on xml Type columns.

index_name

Is the name of the index. Index names must be unique within a table but do not have to be unique within a database. Index names must follow the rules of identifiers.

Primary XML index names cannot start with the following characters: **#**, **##**, **@**, or **@@**.

xml_column_name

Is the **xml** column on which the index is based. Only one **xml** column can be specified in a single XML index definition; however, multiple secondary XML indexes can be created on an **xml** column.

USING XML INDEX xml_index_name

Specifies the primary XML index to use in creating a secondary XML index.

FOR { VALUE | PATH | PROPERTY }

Specifies the type of secondary XML index.

VALUE

Creates a secondary XML index on columns where key columns are (node value and path) of the primary XML index.

PATH

Creates a secondary XML index on columns built on path values and node values in the primary XML index. In the PATH secondary index, the path and node values are key

columns that allow efficient seeks when searching for paths.

PROPERTY

Creates a secondary XML index on columns (PK, path and node value) of the primary XML index where PK is the primary key of the base table.

<object>::=

Is the fully qualified or nonfully qualified object to be indexed.

database_name

Is the name of the database.

schema_name

Is the name of the schema to which the table belongs.

table_name

Is the name of the table to be indexed.

<xml_index_option> ::=

Specifies the options to use when you create the index.

PAD_INDEX = { ON | OFF }

Specifies index padding. The default is OFF.

ON

The percentage of free space that is specified by fillfactor is applied to the intermediate-level pages of the index.

OFF or fillfactor is not specified

The intermediate-level pages are filled to near capacity, leaving sufficient space for at least one row of the maximum size the index can have, considering the set of keys on the intermediate pages.

The PAD_INDEX option is useful only when FILLFACTOR is specified, because PAD_INDEX uses the percentage specified by FILLFACTOR. If the percentage specified for FILLFACTOR is not large enough to allow for one row, the Database Engine internally overrides the percentage to allow for the minimum. The number of rows on an intermediate index page is never less than two, regardless of how low the value of fillfactor.

FILLFACTOR = fillfactor

Specifies a percentage that indicates how full the Database Engine should make the leaf level of each index page during index creation or rebuild. fillfactor must be an integer value from 1 to 100. The default is 0. If fillfactor is 100 or 0, the Database Engine creates indexes with leaf pages filled to capacity.

Note

Fill factor values 0 and 100 are the same in all respects.

The FILLFACTOR setting applies only when the index is created or rebuilt. The Database

Engine does not dynamically keep the specified percentage of empty space in the pages. To view the fill factor setting, use the sys.indexes catalog view.

Important

Creating a clustered index with a FILLFACTOR less than 100 affects the amount of storage space the data occupies because the Database Engine redistributes the data when it creates the clustered index.

For more information, see Fill Factor.

SORT_IN_TEMPDB = { ON | OFF }

Specifies whether to store temporary sort results in **tempdb**. The default is OFF.

ON

The intermediate sort results that are used to build the index are stored in **tempdb**. This may reduce the time required to create an index if **tempdb** is on a different set of disks than the user database. However, this increases the amount of disk space that is used during the index build.

OFF

The intermediate sort results are stored in the same database as the index.

In addition to the space required in the user database to create the index, **tempdb** must have about the same amount of additional space to hold the intermediate sort results. For more information, see tempdb and Index Creation.

IGNORE_DUP_KEY = OFF

Has no effect for XML indexes because the index type is never unique. Do not set this option to ON, or else an error is raised.

DROP_EXISTING = { ON | OFF }

Specifies that the named, preexisting XML index is dropped and rebuilt. The default is OFF.

ON

The existing index is dropped and rebuilt. The index name specified must be the same as a currently existing index; however, the index definition can be modified. For example, you can specify different columns, sort order, partition scheme, or index options.

OFF

An error is displayed if the specified index name already exists.

The index type cannot be changed by using DROP_EXISTING. Also, a primary XML index cannot be redefined as a secondary XML index, or vice versa.

ONLINE = OFF

Specifies that underlying tables and associated indexes are not available for queries and data modification during the index operation. In this version of SQL Server, online index builds are not supported for XML indexes. If this option is set to ON for a XML index, an error is raised. Either omit the ONLINE option or set ONLINE to OFF.

An offline index operation that creates, rebuilds, or drops a XML index, acquires a Schema modification (Sch-M) lock on the table. This prevents all user access to the underlying table for the duration of the operation.

Note

Online index operations are not available in every edition of Microsoft SQL Server. For a list of features that are supported by the editions of SQL Server, see Features Supported by the Editions of SQL Server 2012.

ALLOW_ROW_LOCKS = { ON | OFF }

Specifies whether row locks are allowed. The default is ON.

ON

Row locks are allowed when accessing the index. The Database Engine determines when row locks are used.

OFF

Row locks are not used.

ALLOW_PAGE_LOCKS = { ON | OFF }

Specifies whether page locks are allowed. The default is ON.

ON

Page locks are allowed when accessing the index. The Database Engine determines when page locks are used.

OFF

Page locks are not used.

MAXDOP = max_degree_of_parallelism

Overrides the Configure the max degree of parallelism Server Configuration Option configuration option for the duration of the index operation. Use MAXDOP to limit the number of processors used in a parallel plan execution. The maximum is 64 processors.

Important

Although the MAXDOP option is syntactically supported for all XML indexes, for a primary XML index, CREATE XML INDEX uses only a single processor.

max_degree_of_parallelism can be:

1

Suppresses parallel plan generation.

>1

Restricts the maximum number of processors used in a parallel index operation to the specified number or fewer based on the current system workload.

0 (default)

Uses the actual number of processors or fewer based on the current system workload.

For more information, see Configuring Parallel Index Operations.

Note

Parallel index operations are not available in every edition of Microsoft SQL Server. For a list of features that are supported by the editions of SQL Server, see Features Supported by the Editions of SQL Server 2012.

Remarks

Computed columns derived from **xml** data types can be indexed either as a key or included nonkey column as long as the computed column data type is allowable as an index key column or nonkey column. You cannot create a primary XML index on a computed **xml** column.

To view information about XML indexes, use the sys.xml_indexes catalog view.

For more information about XML indexes, see Indexes on xml Type columns.

Additional Remarks on Index Creation

For more information about index creation, see the "Remarks" section in CREATE INDEX (Transact-SQL).

Examples

A. Creating a primary XML index

The following example creates a primary XML index on the `CatalogDescription` column in the `Production.ProductModel` table.

```
Copy Code
USE AdventureWorks2012;
GO
IF EXISTS (SELECT * FROM sys.indexes
            WHERE name = N'PXML_ProductModel_CatalogDescription')
    DROP INDEX PXML_ProductModel_CatalogDescription
        ON Production.ProductModel;
GO
CREATE PRIMARY XML INDEX PXML_ProductModel_CatalogDescription
    ON Production.ProductModel (CatalogDescription);
GO
```

B. Creating a secondary XML index

The following example creates a secondary XML index on the `CatalogDescription` column in the `Production.ProductModel` table.

```
USE AdventureWorks2012;
```

```
GO
IF EXISTS (SELECT name FROM sys.indexes
            WHERE name = N'IXML_ProductModel_CatalogDescription_Path')
    DROP INDEX IXML_ProductModel_CatalogDescription_Path
        ON Production.ProductModel;
GO
CREATE XML INDEX IXML_ProductModel_CatalogDescription_Path
    ON Production.ProductModel (CatalogDescription)
    USING XML INDEX PXML_ProductModel_CatalogDescription FOR PATH ;
GO
```

See Also

ALTER INDEX (Transact-SQL)
CREATE INDEX (Transact-SQL)
CREATE PARTITION FUNCTION
CREATE PARTITION SCHEME
CREATE SPATIAL INDEX (Transact-SQL)
CREATE STATISTICS
CREATE TABLE
Data Types
DBCC SHOW_STATISTICS
DROP INDEX
Indexes on xml Type columns
sys.indexes
sys.index_columns
sys.xml_indexes
EVENTDATA (Transact-SQL)
Indexes on xml Type columns

CREATE XML SCHEMA COLLECTION

Imports the schema components into a database.

 Transact-SQL Syntax Conventions

Syntax

CREATE XML SCHEMA COLLECTION [**<relational_schema>.]sql_identifier** AS **Expression**

Arguments

relational_schema

Identifies the relational schema name. If not specified, default relational schema is assumed.

sql_identifier

Is the SQL identifier for the XML schema collection.

Expression

Is a string constant or scalar variable. Is **varchar**, **varbinary**, **nvarchar**, or **xml** type.

Remarks

You can also add new namespaces to the collection or add new components to existing namespaces in the collection by using ALTER XML SCHEMA COLLECTION.

To remove collections, use DROP XML SCHEMA COLLECTION (Transact-SQL).

Permissions

To create an XML SCHEMA COLLECTION requires at least one of the following sets of permissions:

- CONTROL permission on the server
- ALTER ANY DATABASE permission on the server
- ALTER permission on the database
- CONTROL permission in the database
- ALTER ANY SCHEMA permission and CREATE XML SCHEMA COLLECTION permission in the database
- ALTER or CONTROL permission on the relational schema and CREATE XML SCHEMA COLLECTION permission in the database

Examples

A. Creating XML schema collection in the database

The following example creates the XML schema collection `ManuInstructionsSchemaCollection`. The collection has only one schema namespace.

```
-- Create a sample database in which to load the XML schema collection.
CREATE DATABASE SampleDB
GO
USE SampleDB
GO
CREATE XML SCHEMA COLLECTION ManuInstructionsSchemaCollection AS
```

```
N'<?xml version="1.0" encoding="UTF-16"?>
<xsd:schema
targetNamespace="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions"
   xmlns          ="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions"
   elementFormDefault="qualified"
   attributeFormDefault="unqualified"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema" >

    <xsd:complexType name="StepType" mixed="true" >
        <xsd:choice  minOccurs="0" maxOccurs="unbounded" >
            <xsd:element name="tool" type="xsd:string" />
            <xsd:element name="material" type="xsd:string" />
            <xsd:element name="blueprint" type="xsd:string" />
            <xsd:element name="specs" type="xsd:string" />
            <xsd:element name="diag" type="xsd:string" />
        </xsd:choice>
    </xsd:complexType>

    <xsd:element  name="root">
        <xsd:complexType mixed="true">
            <xsd:sequence>
                <xsd:element name="Location" minOccurs="1"
maxOccurs="unbounded">
                    <xsd:complexType mixed="true">
                        <xsd:sequence>
                            <xsd:element name="step" type="StepType"
minOccurs="1" maxOccurs="unbounded" />
                        </xsd:sequence>
                        <xsd:attribute name="LocationID" type="xsd:integer"
use="required"/>
                        <xsd:attribute name="SetupHours" type="xsd:decimal"
use="optional"/>
```

```
                        <xsd:attribute name="MachineHours" type="xsd:decimal"
use="optional"/>
                        <xsd:attribute name="LaborHours" type="xsd:decimal"
use="optional"/>
                        <xsd:attribute name="LotSize" type="xsd:decimal"
use="optional"/>
                    </xsd:complexType>
                </xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>' ;
GO
-- Verify - list of collections in the database.
select *
from sys.xml_schema_collections
-- Verify - list of namespaces in the database.
select name
from sys.xml_schema_namespaces

-- Use it. Create a typed xml variable. Note collection name specified.
DECLARE @x xml (ManuInstructionsSchemaCollection)
GO
--Or create a typed xml column.
CREATE TABLE T (
        i int primary key,
        x xml (ManuInstructionsSchemaCollection))
GO
-- Clean up
DROP TABLE T
GO
DROP XML SCHEMA COLLECTION ManuInstructionsSchemaCollection
Go
USE Master
```

```
GO
DROP DATABASE SampleDB
```

Alternatively, you can assign the schema collection to a variable and specify the variable in the `CREATE XML SCHEMA COLLECTION` statement as follows:

```
DECLARE @MySchemaCollection nvarchar(max)
Set @MySchemaCollection  = N' copy the schema collection here'
CREATE XML SCHEMA COLLECTION MyCollection AS @MySchemaCollection
```

The variable in the example is of `nvarchar(max)` type. The variable can also be of **xml** data type, in which case, it is implicitly converted to a string.

For more information, see Viewing Stored XML Schema.

You may store schema collections in an **xml** type column. In this case, to create XML schema collection, perform the following:

1. Retrieve the schema collection from the column by using a SELECT statement and assign it to a variable of **xml** type, or a **varchar** type.
2. Specify the variable name in the CREATE XML SCHEMA COLLECTION statement.

The CREATE XML SCHEMA COLLECTION stores only the schema components that SQL Server understands; everything in the XML schema is not stored in the database. Therefore, if you want the XML schema collection back exactly the way it was supplied, we recommend that you save your XML schemas in a database column or some other folder on your computer.

B. Specifying multiple schema namespaces in a schema collection

You can specify multiple XML schemas when you create an XML schema collection. For example:

```
CREATE XML SCHEMA COLLECTION MyCollection AS N'
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!-- Contents of schema here -->
</xsd:schema>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!-- Contents of schema here -->
</xsd:schema>'
```

The following example creates the XML schema collection `ProductDescriptionSchemaCollection` that includes two XML schema namespaces.

```
CREATE XML SCHEMA COLLECTION ProductDescriptionSchemaCollection AS
'<xsd:schema
targetNamespace="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelWarrAndMain"
    xmlns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelWarrAndMain"
```

```
    elementFormDefault="qualified"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
    <xsd:element name="Warranty"  >
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="WarrantyPeriod" type="xsd:string"  />
                <xsd:element name="Description" type="xsd:string"  />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
 <xs:schema
targetNamespace="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription"
    xmlns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription"
    elementFormDefault="qualified"
    xmlns:mstns="http://tempuri.org/XMLSchema.xsd"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:wm="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain" >
    <xs:import
namespace="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain" />
    <xs:element name="ProductDescription" type="ProductDescription" />
        <xs:complexType name="ProductDescription">
            <xs:sequence>
                <xs:element name="Summary" type="Summary" minOccurs="0" />
            </xs:sequence>
            <xs:attribute name="ProductModelID" type="xs:string" />
            <xs:attribute name="ProductModelName" type="xs:string" />
        </xs:complexType>
        <xs:complexType name="Summary" mixed="true" >
            <xs:sequence>
```

```
                <xs:any processContents="skip"
namespace="http://www.w3.org/1999/xhtml" minOccurs="0" maxOccurs="unbounded"
/>
            </xs:sequence>
        </xs:complexType>
</xs:schema>'
;
GO -- Clean up
DROP XML SCHEMA COLLECTION ProductDescriptionSchemaCollection
GO
```

C. Importing a schema that does not specify a target namespace

If a schema that does not contain a **targetNamespace** attribute is imported in a collection, its components are associated with the empty string target namespace as shown in the following example. Note that not associating one or more schemas imported in the collection causes multiple schema components (potentially unrelated) to be associated with the default empty string namespace.

```
-- Create a collection that contains a schema with no target namespace.
CREATE XML SCHEMA COLLECTION MySampleCollection AS '
<schema xmlns="http://www.w3.org/2001/XMLSchema"  xmlns:ns="http://ns">
<element name="e" type="dateTime"/>
</schema>'
go
-- Query will return the names of all the collections that
--contain a schema with no target namespace.
SELECT sys.xml_schema_collections.name
FROM   sys.xml_schema_collections
JOIN   sys.xml_schema_namespaces
ON     sys.xml_schema_collections.xml_collection_id =
       sys.xml_schema_namespaces.xml_collection_id
WHERE  sys.xml_schema_namespaces.name=''
```

D. Using an XML schema collection and batches

A schema collection cannot be referenced in the same batch where it is created. If you try to reference a collection in the same batch where it was created, you will get an error saying the collection does not exist. The following example works; however, if you remove `GO` and,

therefore, try to reference the XML schema collection to type an `xml` variable in the same batch, it will return an error.

```
CREATE XML SCHEMA COLLECTION mySC AS '
<schema xmlns="http://www.w3.org/2001/XMLSchema">
      <element name="root" type="string"/>
</schema>
'
GO
CREATE TABLE T (Col1 xml (mySC))
GO
```

See Also

ALTER XML SCHEMA COLLECTION (Transact-SQL)
DROP XML SCHEMA COLLECTION (Transact-SQL)
EVENTDATA (Transact-SQL)
Typed vs. Untyped XML
DROP XML SCHEMA COLLECTION (Transact-SQL)
Guidelines and Limitations of XML Schemas on the Server

DISABLE TRIGGER

Disables a trigger.
 Transact-SQL Syntax Conventions

Syntax

DISABLE TRIGGER { [**schema_name** .] **trigger_name** [**,...n**] | ALL }
ON { **object_name** | DATABASE | ALL SERVER } [;]

Arguments

schema_name

Is the name of the schema to which the trigger belongs. schema_name cannot be specified for DDL or logon triggers.

trigger_name

Is the name of the trigger to be disabled.

ALL

Indicates that all triggers defined at the scope of the ON clause are disabled.

Caution

SQL Server creates triggers in databases that are published for merge replication. Specifying ALL in published databases disables these triggers, which disrupts replication. Verify that the current database is not published for merge replication before specifying ALL.

object_name

Is the name of the table or view on which the DML trigger trigger_name was created to execute.

DATABASE

For a DDL trigger, indicates that trigger_name was created or modified to execute with database scope.

ALL SERVER

For a DDL trigger, indicates that trigger_name was created or modified to execute with server scope. ALL SERVER also applies to logon triggers.

Note

This option is not available in a contained database.

Remarks

Triggers are enabled by default when they are created. Disabling a trigger does not drop it. The trigger still exists as an object in the current database. However, the trigger does not fire when any Transact-SQL statements on which it was programmed are executed. Triggers can be re-enabled by using ENABLE TRIGGER. DML triggers defined on tables can be also be disabled or enabled by using ALTER TABLE.

Permissions

To disable a DML trigger, at a minimum, a user must have ALTER permission on the table or view on which the trigger was created.

To disable a DDL trigger with server scope (ON ALL SERVER) or a logon trigger, a user must have CONTROL SERVER permission on the server. To disable a DDL trigger with database scope (ON DATABASE), at a minimum, a user must have ALTER ANY DATABASE DDL TRIGGER permission in the current database.

Examples

A. Disabling a DML trigger on a table

The following example disables trigger `uAddress` that was created on table `Address`.

```
USE AdventureWorks2012;
```

```
GO
DISABLE TRIGGER Person.uAddress ON Person.Address;
GO
```

B. Disabling a DDL trigger

The following example creates a DDL trigger `safety` with database scope, and then disables it.

```
IF EXISTS (SELECT * FROM sys.triggers
    WHERE parent_class = 0 AND name = 'safety')
DROP TRIGGER safety ON DATABASE;
GO
CREATE TRIGGER safety
ON DATABASE
FOR DROP_TABLE, ALTER_TABLE
AS
   PRINT 'You must disable Trigger "safety" to drop or alter tables!'
   ROLLBACK;
GO
DISABLE TRIGGER safety ON DATABASE;
GO
```

C. Disabling all triggers that were defined with the same scope

The following example disables all DDL triggers that were created at the server scope.

```
USE AdventureWorks2012;
GO
DISABLE Trigger ALL ON ALL SERVER;
```

GO

See Also

sys.triggers (Transact-SQL)
ALTER TRIGGER
CREATE TRIGGER
DROP TRIGGER
sys.triggers

DROP Statements

SQL Server Transact-SQL contains the following DROP statements. Use DROP statements to remove existing entities. For example, use DROP TABLE to remove a table from a database.

In this Section

DROP AGGREGATE	DROP FULLTEXT INDEX	DROP SEARCH PROPERTY LIST (Transact-SQL)
DROP APPLICATION ROLE	DROP FULLTEXT STOPLIST	DROP SEQUENCE (Transact-SQL)
DROP ASSEMBLY	DROP FUNCTION	DROP SERVER AUDIT
DROP ASYMMETRIC KEY	DROP INDEX	DROP SERVER AUDIT SPECIFICATION
DROP BROKER PRIORITY	DROP LOGIN	DROP SERVICE
DROP CERTIFICATE	DROP MASTER KEY	DROP SIGNATURE
DROP CONTRACT	DROP MESSAGE TYPE	DROP STATISTICS
DROP CREDENTIAL	DROP PARTITION FUNCTION	DROP SYMMETRIC KEY
DROP CRYPTOGRAPHIC PROVIDER	DROP PARTITION SCHEME	DROP SYNONYM
DROP DATABASE	DROP PROCEDURE	DROP TABLE
DROP DATABASE AUDIT SPECIFICATION	DROP QUEUE	DROP TRIGGER
DROP DATABASE ENCRYPTION KEY	DROP REMOTE SERVICE BINDING	DROP TYPE
DROP DEFAULT	DROP RESOURCE POOL	DROP USER
DROP ENDPOINT	DROP ROLE	DROP VIEW
DROP EVENT NOTIFICATION	DROP ROUTE	DROP WORKLOAD GROUP
DROP EVENT SESSION	DROP RULE	DROP XML SCHEMA COLLECTION
DROP FULLTEXT CATALOG	DROP SCHEMA	

See Also

ALTER Statements (Transact-SQL)

CREATE Statements (Transact-SQL)

DROP AGGREGATE

Removes a user-defined aggregate function from the current database. User-defined aggregate functions are created by using CREATE AGGREGATE.

 Transact-SQL Syntax Conventions

Syntax

DROP AGGREGATE [**schema_name .**] **aggregate_name**

Arguments

schema_name

Is the name of the schema to which the user-defined aggregate function belongs.

aggregate_name

Is the name of the user-defined aggregate function you want to drop.

Remarks

DROP AGGREGATE does not execute if there are any views, functions, or stored procedures created with schema binding that reference the user-defined aggregate function you want to drop.

Permissions

To execute DROP AGGREGATE, at a minimum, a user must have ALTER permission on the schema to which the user-defined aggregate belongs, or CONTROL permission on the aggregate.

Examples

The following example drops the aggregate `Concatenate`.

```
DROP AGGREGATE dbo.Concatenate
```

See Also

Creating User-defined Aggregates

Creating User-Defined Aggregate Functions

DROP APPLICATION ROLE

Removes an application role from the current database.

Transact-SQL Syntax Conventions

Syntax

DROP APPLICATION ROLE **rolename**

Arguments

rolename

Specifies the name of the application role to be dropped.

Remarks

If the application role owns any securables it cannot be dropped. Before dropping an application role that owns securables, you must first transfer ownership of the securables, or drop them.

Caution

Beginning with SQL Server 2005, the behavior of schemas changed. As a result, code that assumes that schemas are equivalent to database users may no longer return correct results. Old catalog views, including , should not be used in a database in which any of the following DDL statements have ever been used: CREATE SCHEMA, ALTER SCHEMA, DROP SCHEMA, CREATE USER, ALTER USER, DROP USER, CREATE ROLE, ALTER ROLE, DROP ROLE, CREATE APPROLE, ALTER APPROLE, DROP APPROLE, ALTER AUTHORIZATION. In such databases you must instead use the new catalog views. The new catalog views take into account the separation of principals and schemas that was introduced in SQL Server 2005. For more information about catalog views, see .

Permissions

Requires ALTER ANY APPLICATION ROLE permission on the database.

Examples

Drop application role "weekly_ledger" from the database.

```
DROP APPLICATION ROLE weekly_ledger;
GO
```

See Also

Application Roles

CREATE APPLICATION ROLE (Transact-SQL)

ALTER APPLICATION ROLE (Transact-SQL)

eventdata (Transact-SQL)

DROP ASSEMBLY

Removes an assembly and all its associated files from the current database. Assemblies are created by using CREATE ASSEMBLY and modified by using ALTER ASSEMBLY.

 Transact-SQL Syntax Conventions

Syntax

DROP ASSEMBLY **assembly_name** [**,...n**]
[WITH NO DEPENDENTS]
[;]

Arguments

assembly_name

Is the name of the assembly you want to drop.

WITH NO DEPENDENTS

If specified, drops only assembly_name and none of the dependent assemblies that are referenced by the assembly. If not specified, DROP ASSEMBLY drops assembly_name and all dependent assemblies.

Remarks

Dropping an assembly removes an assembly and all its associated files, such as source code and debug files, from the database.

If WITH NO DEPENDENTS is not specified, DROP ASSEMBLY drops assembly_name and all dependent assemblies. If an attempt to drop any dependent assemblies fails, DROP ASSEMBLY returns an error.

DROP ASSEMBLY returns an error if the assembly is referenced by another assembly that exists in the database or if it is used by common language runtime (CLR) functions, procedures, triggers, user-defined types or aggregates in the current database.

DROP ASSEMBLY does not interfere with any code referencing the assembly that is currently running. However, after DROP ASSEMBLY executes, any attempts to invoke the assembly code will fail.

Permissions

Requires ownership of the assembly, or CONTROL permission on it.

Examples

The following example assumes the assembly `HelloWorld` is already created in the instance of SQL Server.

```
DROP ASSEMBLY Helloworld
```

See Also

Getting Information About Assemblies
ALTER ASSEMBLY
EVENTDATA (Transact-SQL)
Getting Information About Assemblies

DROP ASYMMETRIC KEY

Removes an asymmetric key from the database.

 Transact-SQL Syntax Conventions

Syntax

DROP ASYMMETRIC KEY **`key_name`** [REMOVE PROVIDER KEY]

Arguments

key_name

Is the name of the asymmetric key to be dropped from the database.

REMOVE PROVIDER KEY

Removes an Extenisble Key Management (EKM) key from an EKM device. For more information about Extensible Key Management, see Understanding Extensible Key Management (EKM).

Remarks

An asymmetric key with which a symmetric key in the database has been encrypted, or to which a user or login is mapped, cannot be dropped. Before you drop such a key, you must drop any user or login that is mapped to the key. You must also drop or change any symmetric key encrypted with the asymmetric key. You can use the DROP ENCRYPTION option of ALTER SYMMETRIC KEY to remove encryption by an asymmetric key.

Metadata of asymmetric keys can be accessed by using the sys.asymmetric_keys catalog view. The keys themselves cannot be directly viewed from inside the database.

If the asymmetric key is mapped to an Extensible Key Management (EKM) key on an EKM device and the REMOVE PROVIDER KEY option is not specified, the key will be dropped from the database but not the device. A warning will be issued.

Permissions

Requires CONTROL permission on the asymmetric key.

Examples

The following example removes the asymmetric key `MirandaXAsymKey6` from the `AdventureWorks2012` database.

```
USE AdventureWorks2012;
```

```
DROP ASYMMETRIC KEY MirandaXAsymKey6;
```

See Also

ALTER SYMMETRIC KEY (Transact-SQL)
ALTER ASYMMETRIC KEY (Transact-SQL)
Encryption Hierarchy
ALTER SYMMETRIC KEY (Transact-SQL)

DROP AVAILABILITY GROUP

Removes the specified availability group and all of its replicas. Dropping an availability group also deletes the associated availability group listener, if any.

 Transact-SQL Syntax Conventions

Syntax

DROP AVAILABILITY GROUP **group_name**
[;]

Arguments

group_name

Specifies the name of the availability group to be dropped.

Limitations and Restrictions

Executing DROP AVAILABILITY GROUP requires that HADR service is enabled on the server instance. For more information, see The HADR Service (SQL Server).

DROP AVAILABILITY GROUP cannot be executed as part of batches or within transactions. Also, expressions and variables are not supported.

You can drop an availability group from any Windows Server Failover Clustering (WSFC) node that possesses the correct security credentials for the availability group. This enables you to delete an availability group when none of its availability replicas remain.

For more information, see Deleting an Availability Group (SQL Server).

Security

Permissions

Requires ALTER AVAILABILITY GROUP permission on the availability group, CONTROL AVAILABILITY GROUP permission, ALTER ANY AVAILABILITY GROUP permission, or CONTROL SERVER permission. To drop an availability group that is not hosted by the local server instance you need CONTROL SERVER permission or CONTROL permission on that availability group.

Examples

The following example drops the `AccountsAG` availability group.

```
DROP AVAILABILITY GROUP AccountsAG;
```

See Also

ALTER AVAILABILITY GROUP (Transact-SQL)

CREATE AVAILABILITY GROUP (Transact-SQL)

Delete an Availability Group (SQL Server)

DROP BROKER PRIORITY

Removes a conversation priority from the current database.

 Transact-SQL Syntax Conventions

Syntax

DROP BROKER PRIORITY **`ConversationPriorityName`**

[;]

Arguments

ConversationPriorityName

Specifies the name of the conversation priority to be removed.

Remarks

When you drop a conversation priority, any existing conversations continue to operate with the priority levels they were assigned from the conversation priority.

Permissions

Permission for creating a conversation priority defaults to members of the db_ddladmin or db_owner fixed database roles, and to the sysadmin fixed server role. Requires ALTER permission on the database.

Examples

The following example drops the conversation priority named `InitiatorAToTargetPriority`.

```
DROP BROKER PRIORITY InitiatorAToTargetPriority;
```

See Also

ALTER BROKER PRIORITY (Transact-SQL)

CREATE BROKER PRIORITY (Transact-SQL)

sys.conversation_priorities (Transact-SQL)

DROP CERTIFICATE

Removes a certificate from the database.

 Transact-SQL Syntax Conventions

Syntax

```
DROP CERTIFICATE certificate_name
```

Arguments

certificate_name

Is the unique name by which the certificate is known in the database.

Remarks

Certificates can only be dropped if no entities are associated with them.

Permissions

Requires CONTROL permission on the certificate.

Examples

The following example drops the certificate `Shipping04` from the `AdventureWorks` database.

```
USE AdventureWorks2012;
DROP CERTIFICATE Shipping04;
```

See Also

EVENTDATA (Transact-SQL)
CREATE CERTIFICATE (Transact-SQL)
ALTER CERTIFICATE (Transact-SQL)
Encryption Hierarchy
EVENTDATA (Transact-SQL)

DROP CONTRACT

Drops an existing contract from a database.

 Transact-SQL Syntax Conventions

Syntax

```
DROP CONTRACT contract_name
[ ; ]
```

Arguments

contract_name

The name of the contract to drop. Server, database, and schema names cannot be specified.

Remarks

You cannot drop a contract if any services or conversation priorities refer to the contract.

When you drop a contract, Service Broker ends any existing conversations that use the contract with an error.

Permissions

Permission for dropping a contract defaults to the owner of the contract, members of the db_ddladmin or db_owner fixed database roles, and members of the sysadmin fixed server role.

Examples

The following example removes the contract `//Adventure-Works.com/Expenses/ExpenseSubmission` from the database.

```
DROP CONTRACT
    [//Adventure-Works.com/Expenses/ExpenseSubmission] ;
```

See Also

ALTER BROKER PRIORITY (Transact-SQL)

ALTER SERVICE (Transact-SQL)

CREATE CONTRACT

DROP BROKER PRIORITY (Transact-SQL)

DROP SERVICE (Transact-SQL)

EVENTDATA (Transact-SQL)

DROP CREDENTIAL

Removes a credential from the server.

 Transact-SQL Syntax Conventions

Syntax

DROP CREDENTIAL **`credential_name`**

Arguments

credential_name

Is the name of the credential to remove from the server.

Remarks

To drop the secret associated with a credential without dropping the credential itself, use ALTER CREDENTIAL.

Information about credentials is visible in the **sys.credentials** catalog view.

Permissions

Requires ALTER ANY CREDENTIAL permission. If dropping a system credential, requires CONTROL SERVER permission.

Examples

The following example removes the credential called `Saddles`.

```
DROP CREDENTIAL Saddles;
GO
```

See Also

sys.credentials (Transact-SQL)

CREATE CREDENTIAL (Transact-SQL)

ALTER CREDENTIAL (Transact-SQL)

sys.credentials (Transact-SQL)

DROP CRYPTOGRAPHIC PROVIDER

Drops a cryptographic provider within SQL Server.

 Transact-SQL Syntax Conventions

Syntax

DROP CRYPTOGRAPHIC PROVIDER **`provider_name`**

Arguments

provider_name

Is the name of the Extensible Key Management provider.

Remarks

To delete an Extensible Key Management (EKM) provider, all sessions that use the provider must be stopped.

An EKM provider can only be dropped if there are no credentials mapped to it.

If there are keys mapped to an EKM provider when it is dropped the GUIDs for the keys remain stored in SQL Server. If a provider is created later with the same key GUIDs, the keys will be reused.

Permissions

Requires CONTROL permission on the symmetric key.

Examples

The following example drops a cryptographic provider called `SecurityProvider`.

```
/* First, disable provider to perform the upgrade.
This will terminate all open cryptographic sessions. */
ALTER CRYPTOGRAPHIC PROVIDER SecurityProvider
SET ENABLED = OFF;
GO
/* Drop the provider. */
DROP CRYPTOGRAPHIC PROVIDER SecurityProvider;
GO
```

See Also

Understanding Extensible Key Management (EKM)
CREATE CRYPTOGRAPHIC PROVIDER (Transact-SQL)
ALTER CRYPTOGRAPHIC PROVIDER (Transact-SQL)
CREATE SYMMETRIC KEY (Transact-SQL)

DROP DATABASE

Removes one or more databases or database snapshots from an instance of SQL Server.

 Transact-SQL Syntax Conventions

Syntax

```
DROP DATABASE { database_name | database_snapshot_name } [ ,...n ]
[;]
```

Arguments

database_name

Specifies the name of the database to be removed. To display a list of databases, use the sys.databases catalog view.

database_snapshot_name

Specifies the name of a database snapshot to be removed.

Remarks

To use DROP DATABASE, the database context of the connection cannot be the same as the database or database snapshot to be dropped.

The DROP DATABASE statement must run in autocommit mode and is not allowed in an explicit or implicit transaction. Autocommit mode is the default transaction management mode.

Dropping a Database

System databases cannot be dropped.

Dropping a database deletes the database from an instance of SQL Server and deletes the physical disk files used by the database. If the database or any one of its files is offline when it is dropped, the disk files are not deleted. These files can be deleted manually by using Windows Explorer. To remove a database from the current server without deleting the files from the file system, use sp_detach_db.

You cannot drop a database currently being used. This means open for reading or writing by any user. To remove users from the database, use ALTER DATABASE to set the database to SINGLE_USER.

Any database snapshots on a database must be dropped before the database can be dropped.

If the database is involved in log shipping, remove log shipping before dropping the database. For more information, see Log Shipping Overview.

A database can be dropped regardless of its state: offline, read-only, suspect, and so on. To display the current state of a database, use the **sys.databases** catalog view.

A dropped database can be re-created only by restoring a backup. Database snapshots cannot be backed up and, therefore, cannot be restored.

When a database is dropped, the master database should be backed up.

Dropping a Database Snapshot

Dropping a database snapshot deletes the database snapshot from an instance of SQL Server and deletes the physical NTFS File System sparse files used by the snapshot. For information about using sparse files by database snapshots, see Database Snapshots (SQL Server).

Dropping a database snapshot clears the plan cache for the instance of SQL Server. Clearing the plan cache causes a recompilation of all subsequent execution plans and can cause a sudden, temporary decrease in query performance. For each cleared cachestore in the plan cache, the SQL Server error log contains the following informational message: "SQL Server has encountered %d occurrence(s) of cachestore flush for the '%s' cachestore (part of plan cache) due to some database maintenance or reconfigure operations". This message is logged every five minutes as long as the cache is flushed within that time interval.

Dropping a Database Used in Replication

To drop a database published for transactional replication, or published or subscribed to merge replication, you must first remove replication from the database. If a database is damaged or replication cannot first be removed or both, in most cases you still can drop the database by using ALTER DATABASE to set the database offline and then dropping it.

Permissions

Requires the **CONTROL** permission on the database, or **ALTER ANY DATABASE** permission, or membership in the **db_owner** fixed database role.

Examples

A. Dropping a single database

The following example removes the `Sales` database.

```
DROP DATABASE Sales;
```

B. Dropping multiple databases

The following example removes each of the listed databases.

```
DROP DATABASE Sales, NewSales;
```

C. Dropping a database snapshot

The following example drops a database snapshot, named `sales_snapshot0600`, without affecting the source database.

```
DROP DATABASE sales_snapshot0600;
```

See Also

ALTER DATABASE

CREATE DATABASE

EVENTDATA (Transact-SQL)

sys.databases (Transact-SQL)

DROP DATABASE AUDIT SPECIFICATION

Drops a database audit specification object using the SQL Server Audit feature. For more information, see Understanding SQL Server Audit.

 Transact-SQL Syntax Conventions

Syntax

DROP DATABASE AUDIT SPECIFICATION **audit_specification_name**

[;]

Arguments

audit_specification_name

Name of an existing audit specification object.

Remarks

A DROP DATABASE AUDIT SPECIFICATION removes the metadata for the audit specification, but not the audit data collected before the DROP command was issued. You must set the state of a

database audit specification to OFF using `ALTER DATABASE AUDIT SPECIFICATION` before it can be dropped.

Permissions

Users with the **ALTER ANY DATABASE AUDIT** permission can drop database audit specifications.

Examples

A. Dropping a Database Audit Specification

The following example drops an audit called `HIPAA_Audit_DB_Specification`.

```
DROP DATABASE AUDIT SPECIFICATION HIPAA_Audit_DB_Specification;
GO
```

For a full example of creating an audit, see Understanding SQL Server Audit.

See Also

CREATE SERVER AUDIT (Transact-SQL)
ALTER SERVER AUDIT (Transact-SQL)
DROP SERVER AUDIT (Transact-SQL)
CREATE SERVER AUDIT SPECIFICATION (Transact-SQL)
ALTER SERVER AUDIT SPECIFICATION (Transact-SQL)
DROP SERVER AUDIT SPECIFICATION (Transact-SQL)
CREATE DATABASE AUDIT SPECIFICATION (Transact-SQL)
ALTER DATABASE AUDIT SPECIFICATION (Transact-SQL)
ALTER AUTHORIZATION (Transact-SQL)
fn_get_audit_file (Transact-SQL)
sys.server_audits (Transact-SQL)
sys.server_file_audits (Transact-SQL)
sys.server_audit_specifications (Transact-SQL)
sys.server_audit_specifications_details (Transact-SQL)
sys.database_ audit_specifications (Transact-SQL)
sys.database_audit_specification_details (Transact-SQL)
sys.dm_server_audit_status
sys.dm_audit_actions
sys.dm_audit_class_type_map
Create a Server Audit and Server Audit Specification

DROP DATABASE ENCRYPTION KEY

Drops a database encryption key that is used in transparent database encryption. For more information about transparent database encryption, see Understanding Transparent Data Encryption (TDE).

 Transact-SQL Syntax Conventions

Syntax

```
DROP DATABASE ENCRYPTION KEY
```

Remarks

If the database is encrypted, you must first remove encryption from the database by using the ALTER DATABASE statement. Wait for decryption to complete before removing the database encryption key. For more information about the ALTER DATABASE statement, see ALTER DATABASE SET Options (Transact-SQL). To view the state of the database, use the sys.dm_database_encryption_keys dynamic management view.

Permissions

Requires CONTROL permission on the database.

Examples

The following example removes the database encryption and drops the database encryption key.

```
ALTER DATABASE AdventureWorks2012;
SET ENCRYPTION OFF;
GO
/* Wait for decryption operation to complete, look for a
value of  1 in the query below. */
SELECT encryption_state
FROM sys.dm_database_encryption_keys;
GO
USE AdventureWorks2012;
GO
DROP DATABASE ENCRYPTION KEY;
GO
```

See Also

Understanding Transparent Data Encryption (TDE)

SQL Server Encryption

SQL Server and Database Encryption Keys (Database Engine)
Encryption Hierarchy
ALTER DATABASE SET Options (Transact-SQL)
CREATE DATABASE ENCRYPTION KEY (Transact-SQL)
ALTER DATABASE ENCRYPTION KEY (Transact-SQL)
sys.dm_database_encryption_keys

DROP DEFAULT

Removes one or more user-defined defaults from the current database.

Important
DROP DEFAULT will be removed in the next version of Microsoft SQL Server. Do not use DROP DEFAULT in new development work, and plan to modify applications that currently use them. Instead, use default definitions that you can create by using the DEFAULT keyword of ALTER TABLE or CREATE TABLE.

Transact-SQL Syntax Conventions

Syntax

```
DROP DEFAULT { [ schema_name . ] default_name } [ ,...n ] [ ; ]
```

Arguments

schema_name

Is the name of the schema to which the default belongs.

default_name

Is the name of an existing default. To see a list of defaults that exist, execute **sp_help**.
Defaults must comply with the rules for identifiers. Specifying the default schema name is optional.

Remarks

Before dropping a default, unbind the default by executing **sp_unbindefault** if the default is currently bound to a column or an alias data type.

After a default is dropped from a column that allows for null values, NULL is inserted in that position when rows are added and no value is explicitly supplied. After a default is dropped from a NOT NULL column, an error message is returned when rows are added and no value is explicitly supplied. These rows are added later as part of the typical INSERT statement behavior.

Permissions

To execute DROP DEFAULT, at a minimum, a user must have ALTER permission on the schema to which the default belongs.

Examples

A. Dropping a default

If a default has not been bound to a column or to an alias data type, it can just be dropped using DROP DEFAULT. The following example removes the user-created default named `datedflt`.

```
USE AdventureWorks2012;
GO
IF EXISTS (SELECT name FROM sys.objects
         WHERE name = 'datedflt'
            AND type = 'D')
   DROP DEFAULT datedflt
GO
```

B. Dropping a default that has been bound to a column

The following example unbinds the default associated with the `EmergencyContactPhone` column of the `Contact` table and then drops the default named `phonedflt`.

```
USE AdventureWorks2012;
GO
IF EXISTS (SELECT name FROM sys.objects
         WHERE name = 'phonedflt'
            AND type = 'D')
   BEGIN
      EXEC sp_unbindefault 'Person.Contact.Phone'
      DROP DEFAULT phonedflt
   END
GO
```

See Also

CREATE DEFAULT
sp_helptext
sp_help
sp_unbindefault

DROP ENDPOINT

Drops an existing endpoint.

 Transact-SQL Syntax Conventions

Syntax

DROP ENDPOINT **endPointName**

Arguments

endPointName

Is the name of the endpoint to be removed.

Remarks

The ENDPOINT DDL statements cannot be executed inside a user transaction.

Permissions

User must be a member of the **sysadmin** fixed server role, the owner of the endpoint, or have been granted CONTROL permission on the endpoint.

Examples

The following example removes a previously created endpoint called `sql_endpoint`.

```
DROP ENDPOINT sql_endpoint;
```

See Also

CREATE ENDPOINT (Transact-SQL)
ALTER ENDPOINT (Transact-SQL)
EVENTDATA (Transact-SQL)

DROP EVENT NOTIFICATION

Removes an event notification trigger from the current database.

 Transact-SQL Syntax Conventions

Syntax

DROP EVENT NOTIFICATION **notification_name** [**,...n**]
ON { SERVER | DATABASE | QUEUE **queue_name** }
[;]

Arguments

notification_name

Is the name of the event notification to remove. Multiple event notifications can be specified.
To see a list of currently created event notifications, use sys.events (Transact-SQL).

SERVER

Indicates the scope of the event notification applies to the current server. SERVER must be specified if it was specified when the event notification was created.

DATABASE

Indicates the scope of the event notification applies to the current database. DATABASE must be specified if it was specified when the event notification was created.

QUEUE queue_name

Indicates the scope of the event notification applies to the queue specified by queue_name. QUEUE must be specified if it was specified when the event notification was created. queue_name is the name of the queue and must also be specified.

Remarks

If an event notification fires within a transaction and is dropped within the same transaction, the event notification instance is sent, and then the event notification is dropped.

Permissions

To drop an event notification that is scoped at the database level, at a minimum, a user must be the owner of the event notification or have ALTER ANY DATABASE EVENT NOTIFICATION permission in the current database.

To drop an event notification that is scoped at the server level, at a minimum, a user must be the owner of the event notification or have ALTER ANY EVENT NOTIFICATION permission in the server.

To drop an event notification on a specific queue, at a minimum, a user must be the owner of the event notification or have ALTER permission on the parent queue.

Examples

The following example creates a database-scoped event notification, then drops it:

```
USE AdventureWorks2012;
GO
CREATE EVENT NOTIFICATION NotifyALTER_T1
ON DATABASE
FOR ALTER_TABLE
TO SERVICE 'NotifyService',
    '8140a771-3c4b-4479-8ac0-81008ab17984';
GO
DROP EVENT NOTIFICATION NotifyALTER_T1
ON DATABASE;
```

See Also

CREATE EVENT NOTIFICATION

EVENTDATA (Transact-SQL)

sys.event_notifications (Transact-SQL)

sys.events (Transact-SQL)

DROP EVENT SESSION

Drops an event session.

 Transact-SQL Syntax Conventions

Syntax

```
DROP EVENT SESSION event_session_name
ON SERVER
```

Arguments

event_session_name

Is the name of an existing event session.

Remarks

When you drop an event session, all configuration information, such as targets and session parameters, is completely removed.

Permissions

Requires the ALTER ANY EVENT SESSION permission.

Examples

The following example shows how to drop an event session.

```
DROP EVENT SESSION evt_spin_lock_diagnosis
ON SERVER
```

See Also

CREATE EVENT SESSION (Transact-SQL)

ALTER EVENT SESSION (Transact-SQL)

sys.server_event_sessions

DROP FULLTEXT CATALOG

Removes a full-text catalog from a database. You must drop all full-text indexes associated with the catalog before you drop the catalog.

 Transact-SQL Syntax Conventions

Syntax

DROP FULLTEXT CATALOG **`catalog_name`**

Arguments

catalog_name

Is the name of the catalog to be removed. If catalog_name does not exist, Microsoft SQL Server returns an error and does not perform the DROP operation. The filegroup of the full-text catalog must not be marked OFFLINE or READONLY for the command to succeed.

Permissions

User must have DROP permission on the full-text catalog or be a member of the **db_owner**, or **db_ddladmin** fixed database roles.

See Also

sys.fulltext_catalogs (Transact-SQL)

_Full-Text Search

CREATE FULLTEXT CATALOG

Full-Text Search

DROP FULLTEXT INDEX

Removes a full-text index from a specified table or indexed view.

 Transact-SQL Syntax Conventions

Syntax

DROP FULLTEXT INDEX ON **`table_name`**

Arguments

table_name

Is the name of the table or indexed view containing the full-text index to be removed.

Remarks

You do not need to drop all columns from the full-text index before using the DROP FULLTEXT INDEX command.

Permissions

The user must have ALTER permission on the table or indexed view, or be a member of the **sysadmin** fixed server role, or **db_owner** or **db_ddladmin** fixed database roles.

Examples

The following example drops the full-text index that exists on the `JobCandidate` table.

```
USE AdventureWorks;
GO
DROP FULLTEXT INDEX ON HumanResources.JobCandidate;
GO
```

See Also

sys.fulltext_indexes (Transact-SQL)
Full-Text Search
CREATE FULLTEXT INDEX
Full-Text Search

DROP FULLTEXT STOPLIST

Drops a full-text stoplist from the database.

 Transact-SQL Syntax Conventions

 **Important**
CREATE FULLTEXT STOPLIST is supported only for compatibility level 100. For compatibility levels 80 and 90, the system stoplist is always assigned to the database.

Syntax

```
DROP FULLTEXT STOPLIST stoplist_name
;
```

Arguments

stoplist_name

Is the name of the full-text stoplist to drop from the database.

Remarks

DROP FULLTEXT STOPLIST fails if any full-text indexes refer to the full-text stoplist being dropped.

Permissions

To drop a stoplist requires having DROP permission on the stoplist or membership in the **db_owner** or **db_ddladmin** fixed database roles.

Examples

The following example drops a full-text stoplist named `myStoplist`.

```
DROP FULLTEXT STOPLIST myStoplist;
```

See Also

ALTER FULLTEXT STOPLIST (Transact-SQL)

CREATE FULLTEXT STOPLIST (Transact-SQL)

sys.fulltext_stoplists (Transact-SQL)

sys.fulltext_stopwords (Transact-SQL)

DROP FUNCTION

Removes one or more user-defined functions from the current database. User-defined functions are created by using CREATE FUNCTION and modified by using ALTER FUNCTION.

 Transact-SQL Syntax Conventions

Syntax

```
DROP FUNCTION { [ schema_name. ] function_name } [ ,...n ]
```

Arguments

schema_name

Is the name of the schema to which the user-defined function belongs.

function_name

Is the name of the user-defined function or functions to be removed. Specifying the schema name is optional. The server name and database name cannot be specified.

Remarks

DROP FUNCTION will fail if there are Transact-SQL functions or views in the database that reference this function and were created by using SCHEMABINDING, or if there are computed columns, CHECK constraints, or DEFAULT constraints that reference the function.

DROP FUNCTION will fail if there are computed columns that reference this function and have been indexed.

Permissions

To execute DROP FUNCTION, at a minimum, a user must have ALTER permission on the schema to which the function belongs, or CONTROL permission on the function.

Examples

A. Dropping a function

The following example drops the `fn_SalesByStore` user-defined function from the `Sales` schema in the `AdventureWorks` sample database. To create this function, see Example B in User-defined Functions (Database Engine).

```
USE AdventureWorks2012;
GO
IF OBJECT_ID (N'Sales.fn_SalesByStore', N'IF') IS NOT NULL
    DROP FUNCTION Sales.fn_SalesByStore;
GO
```

See Also

ALTER FUNCTION

CREATE FUNCTION

OBJECT_ID (Transact-SQL)

EVENTDATA (Transact-SQL)

sys.sql_modules

sys.parameters

DROP INDEX

Important

The syntax defined in `<drop_backward_compatible_index>` will be removed in a future version of Microsoft SQL Server. Avoid using this syntax in new development work, and plan to modify applications that currently use the feature. Use the syntax specified under `<drop_relational_or_xml_index>` instead. XML indexes cannot be dropped using backward compatible syntax.

Removes one or more relational, spatial, filtered, or XML indexes from the current database. You can drop a clustered index and move the resulting table to another filegroup or partition scheme in a single transaction by specifying the MOVE TO option.

The DROP INDEX statement does not apply to indexes created by defining PRIMARY KEY or UNIQUE constraints. To remove the constraint and corresponding index, use ALTER TABLE with the DROP CONSTRAINT clause.

Transact-SQL Syntax Conventions

Syntax

```
DROP INDEX
{ <drop_relational_or_xml_or_spatial_index> [ ,...n ]
| <drop_backward_compatible_index> [ ,...n ]
}
```

<drop_relational_or_xml_or_spatial_index> ::=

index_name ON <object>
[WITH **(** <drop_clustered_index_option> [**,...n**] **)**]

<drop_backward_compatible_index> ::=

[**owner_name.**] **table_or_view_name.index_name**

<object> ::=

{

[**database_name.** [**schema_name**] **.** | **schema_name.**]

table_or_view_name

}

<drop_clustered_index_option> ::=

{

MAXDOP = **max_degree_of_parallelism**

| ONLINE = { ON | OFF }

| MOVE TO { **partition_scheme_name (column_name)**

| **filegroup_name**

| "default"

}

[FILESTREAM_ON { **partition_scheme_name**

| **filestream_filegroup_name**

| "default" }]

}

Arguments

index_name

Is the name of the index to be dropped.

database_name

Is the name of the database.

schema_name

Is the name of the schema to which the table or view belongs.

table_or_view_name

Is the name of the table or view associated with the index. Spatial indexes are supported only on tables.

To display a report of the indexes on an object, use the sys.indexes catalog view.

<drop_clustered_index_option>

Controls clustered index options. These options cannot be used with other index types.

MAXDOP = max_degree_of_parallelism

Overrides the **max degree of parallelism** configuration option for the duration of the index operation. For more information, see Configure the max degree of parallelism Server Configuration Option. Use MAXDOP to limit the number of processors used in a parallel plan execution. The maximum is 64 processors.

Important

MAXDOP is not allowed for spatial indexes or XML indexes.

max_degree_of_parallelism can be:

1

Suppresses parallel plan generation.

>1

Restricts the maximum number of processors used in a parallel index operation to the specified number.

0 (default)

Uses the actual number of processors or fewer based on the current system workload.

For more information, see Configuring Parallel Index Operations.

Note

Parallel index operations not available in every edition of Microsoft SQL Server. For a list of features that are supported by the editions of SQL Server, see Features Supported by the Editions of SQL Server 2012.

ONLINE = ON | OFF

Specifies whether underlying tables and associated indexes are available for queries and data modification during the index operation. The default is OFF.

ON

Long-term table locks are not held. This allows queries or updates to the underlying table to continue.

OFF

Table locks are applied and the table is unavailable for the duration of the index operation.

The ONLINE option can only be specified when you drop clustered indexes. For more information, see the Remarks section.

Note

Online index operations are not available in every edition of Microsoft SQL Server. For a list of features that are supported by the editions of SQL Server, see Features Supported by the Editions of

SQL Server 2012.

MOVE TO { partition_scheme_name (column_name) | filegroup_name | "default"

Specifies a location to move the data rows that currently are in the leaf level of the clustered index. The data is moved to the new location in the form of a heap. You can specify either a partition scheme or filegroup as the new location, but the partition scheme or filegroup must already exist. MOVE TO is not valid for indexed views or nonclustered indexes. If a partition scheme or filegroup is not specified, the resulting table will be located in the same partition scheme or filegroup as was defined for the clustered index.

If a clustered index is dropped by using MOVE TO, any nonclustered indexes on the base table are rebuilt, but they remain in their original filegroups or partition schemes. If the base table is moved to a different filegroup or partition scheme, the nonclustered indexes are not moved to coincide with the new location of the base table (heap). Therefore, even if the nonclustered indexes were previously aligned with the clustered index, they might no longer be aligned with the heap. For more information about partitioned index alignment, see Partitioned Tables and Indexes.

partition_scheme_name (column_name)

Specifies a partition scheme as the location for the resulting table. The partition scheme must have already been created by executing either CREATE PARTITION SCHEME or ALTER PARTITION SCHEME. If no location is specified and the table is partitioned, the table is included in the same partition scheme as the existing clustered index.

The column name in the scheme is not restricted to the columns in the index definition. Any column in the base table can be specified.

filegroup_name

Specifies a filegroup as the location for the resulting table. If no location is specified and the table is not partitioned, the resulting table is included in the same filegroup as the clustered index. The filegroup must already exist.

"default"

Specifies the default location for the resulting table.

Note

In this context, default is not a keyword. It is an identifier for the default filegroup and must be delimited, as in MOVE TO **"**default**"** or MOVE TO **[**default**]**. If **"**default**"** is specified, the QUOTED_IDENTIFIER option must be set ON for the current session. This is the default setting. For more information, see SET QUOTED_IDENTIFIER (Transact-SQL).

FILESTREAM_ON { partition_scheme_name | filestream_filegroup_name | "default" }

Specifies a location to move the FILESTREAM table that currently is in the leaf level of the clustered index. The data is moved to the new location in the form of a heap. You can specify either a partition scheme or filegroup as the new location, but the partition scheme or filegroup must already exist. FILESTREAM ON is not valid for indexed views or nonclustered indexes. If a partition scheme is not specified, the data will be located in the same partition

scheme as was defined for the clustered index.

partition_scheme_name

Specifies a partition scheme for the FILESTREAM data. The partition scheme must have already been created by executing either CREATE PARTITION SCHEME or ALTER PARTITION SCHEME. If no location is specified and the table is partitioned, the table is included in the same partition scheme as the existing clustered index.

If you specify a partition scheme for MOVE TO, you must use the same partition scheme for FILESTREAM ON.

filestream_filegroup_name

Specifies a FILESTREAM filegroup for FILESTREAM data. If no location is specified and the table is not partitioned, the data is included in the default FILESTREAM filegroup.

"default"

Specifies the default location for the FILESTREAM data.

Note

In this context, default is not a keyword. It is an identifier for the default filegroup and must be delimited, as in MOVE TO **"**default**"** or MOVE TO **[**default**]**. If "default" is specified, the QUOTED_IDENTIFIER option must be ON for the current session. This is the default setting. For more information, see SET QUOTED_IDENTIFIER (Transact-SQL).

Remarks

When a nonclustered index is dropped, the index definition is removed from metadata and the index data pages (the B-tree) are removed from the database files. When a clustered index is dropped, the index definition is removed from metadata and the data rows that were stored in the leaf level of the clustered index are stored in the resulting unordered table, a heap. All the space previously occupied by the index is regained. This space can then be used for any database object.

An index cannot be dropped if the filegroup in which it is located is offline or set to read-only.

When the clustered index of an indexed view is dropped, all nonclustered indexes and auto-created statistics on the same view are automatically dropped. Manually created statistics are not dropped.

The syntax table_or_view_name**.**index_name is maintained for backward compatibility. An XML index or spatial index cannot be dropped by using the backward compatible syntax.

When indexes with 128 extents or more are dropped, the Database Engine defers the actual page deallocations, and their associated locks, until after the transaction commits.

Sometimes indexes are dropped and re-created to reorganize or rebuild the index, such as to apply a new fill factor value or to reorganize data after a bulk load. To do this, using ALTER INDEXis more efficient, especially for clustered indexes. ALTER INDEX REBUILD has optimizations to prevent the overhead of rebuilding the nonclustered indexes.

Using Options with DROP INDEX

You can set the following index options when you drop a clustered index: MAXDOP, ONLINE, and MOVE TO.

Use MOVE TO to drop the clustered index and move the resulting table to another filegroup or partition scheme in a single transaction.

When you specify ONLINE = ON, queries and modifications to the underlying data and associated nonclustered indexes are not blocked by the DROP INDEX transaction. Only one clustered index can be dropped online at a time. For a complete description of the ONLINE option, see CREATE INDEX (Transact-SQL).

You cannot drop a clustered index online if the index is disabled on a view, or contains **text**, **ntext**, **image**, **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, or **xml** columns in the leaf-level data rows.

Using the ONLINE = ON and MOVE TO options requires additional temporary disk space.

After an index is dropped, the resulting heap appears in the **sys.indexes** catalog view with NULL in the **name** column. To view the table name, join **sys.indexes** with **sys.tables** on **object_id**. For an example query, see example D.

On multiprocessor computers that are running SQL Server 2005 Enterprise Edition or later, DROP INDEX may use more processors to perform the scan and sort operations associated with dropping the clustered index, just like other queries do. You can manually configure the number of processors that are used to run the DROP INDEX statement by specifying the MAXDOP index option. For more information, see Configuring Parallel Index Operations.

When a clustered index is dropped, the corresponding heap partitions retain their data compression setting unless the partitioning scheme is modified. If the partitioning scheme is changed, all partitions are rebuilt to an uncompressed state (DATA_COMPRESSION = NONE). To drop a clustered index and change the partitioning scheme requires the following two steps:

1. Drop the clustered index.
2. Modify the table by using an ALTER TABLE ... REBUILD ... option specifying the compression option.

When a clustered index is dropped OFFLINE, only the upper levels of clustered indexes are removed; therefore, the operation is quite fast. When a clustered index is dropped ONLINE, SQL Server rebuilds the heap two times, once for step 1 and once for step 2. For more information about data compression, see Creating Compressed Tables and indexes.

XML Indexes

Options cannot be specified when you drop an XML index. Also, you cannot use the table_or_view_name**.**index_name syntax. When a primary XML index is dropped, all associated secondary XML indexes are automatically dropped. For more information, see Indexes on xml Type columns.

Spatial Indexes

Spatial indexes are supported only on tables. When you drop a spatial index, you cannot specify any options or use **.**index_name. The correct syntax is as follows:

DROP INDEX spatial_index_name ON spatial_table_name;

For more information about spatial indexes, see Working with Spatial Indexes.

Permissions

To execute DROP INDEX, at a minimum, ALTER permission on the table or view is required. This permission is granted by default to the **sysadmin** fixed server role and the **db_ddladmin** and **db_owner** fixed database roles.

Examples

A. Dropping an index

The following example deletes the index `IX_ProductVendor_VendorID` on the `ProductVendor` table.

```
USE AdventureWorks2012;
GO
DROP INDEX IX_ProductVendor_BusinessEntityID
    ON Purchasing.ProductVendor;
GO
```

B. Dropping multiple indexes

The following example deletes two indexes in a single transaction.

```
USE AdventureWorks2012;
GO
DROP INDEX
    IX_PurchaseOrderHeader_EmployeeID ON Purchasing.PurchaseOrderHeader,
    IX_Address_StateProvinceID ON Person.Address;
GO
```

C. Dropping a clustered index online and setting the MAXDOP option

The following example deletes a clustered index with the `ONLINE` option set to `ON` and `MAXDOP` set to `8`. Because the MOVE TO option was not specified, the resulting table is stored in the same filegroup as the index.

Note

This example can be executed only in SQL Server 2005 Enterprise Edition or later.

```
USE AdventureWorks2012;
GO
DROP INDEX AK_BillOfMaterials_ProductAssemblyID_ComponentID_StartDate
    ON Production.BillOfMaterials WITH (ONLINE = ON, MAXDOP = 2);
GO
```

D. Dropping a clustered index online and moving the table to a new filegroup

The following example deletes a clustered index online and moves the resulting table (heap) to the filegroup `NewGroup` by using the `MOVE TO` clause. The `sys.indexes`, `sys.tables`, and `sys.filegroups` catalog views are queried to verify the index and table placement in the filegroups before and after the move.

```
USE AdventureWorks2012;
GO
--Create a clustered index on the PRIMARY filegroup if the index does not
exist.
IF NOT EXISTS (SELECT name FROM sys.indexes WHERE name =
            N'AK_BillOfMaterials_ProductAssemblyID_ComponentID_StartDate')
    CREATE UNIQUE CLUSTERED INDEX
        AK_BillOfMaterials_ProductAssemblyID_ComponentID_StartDate
    ON Production.BillOfMaterials (ProductAssemblyID, ComponentID,
        StartDate)
    ON 'PRIMARY';
GO
-- Verify filegroup location of the clustered index.
SELECT t.name AS [Table Name], i.name AS [Index Name], i.type_desc,
    i.data_space_id, f.name AS [Filegroup Name]
FROM sys.indexes AS i
    JOIN sys.filegroups AS f ON i.data_space_id = f.data_space_id
    JOIN sys.tables as t ON i.object_id = t.object_id
        AND i.object_id = OBJECT_ID(N'Production.BillOfMaterials','U')
GO
--Create filegroup NewGroup if it does not exist.
IF NOT EXISTS (SELECT name FROM sys.filegroups
                WHERE name = N'NewGroup')
    BEGIN
    ALTER DATABASE AdventureWorks2012
        ADD FILEGROUP NewGroup;
    ALTER DATABASE AdventureWorks2012
        ADD FILE (NAME = File1,
            FILENAME = 'C:\Program Files\Microsoft SQL
Server\MSSQL10.MSSQLSERVER\MSSQL\DATA\File1.ndf')
        TO FILEGROUP NewGroup;
```

```
    END
GO
--Verify new filegroup
SELECT * from sys.filegroups;
GO
-- Drop the clustered index and move the BillOfMaterials table to
-- the Newgroup filegroup.
-- Set ONLINE = OFF to execute this example on editions other than Enterprise
Edition.
DROP INDEX AK_BillOfMaterials_ProductAssemblyID_ComponentID_StartDate
    ON Production.BillOfMaterials
    WITH (ONLINE = ON, MOVE TO NewGroup);
GO
-- Verify filegroup location of the moved table.
SELECT t.name AS [Table Name], i.name AS [Index Name], i.type_desc,
    i.data_space_id, f.name AS [Filegroup Name]
FROM sys.indexes AS i
    JOIN sys.filegroups AS f ON i.data_space_id = f.data_space_id
    JOIN sys.tables as t ON i.object_id = t.object_id
        AND i.object_id = OBJECT_ID(N'Production.BillOfMaterials','U');
GO
```

E. Dropping a PRIMARY KEY constraint online

Indexes that are created as the result of creating PRIMARY KEY or UNIQUE constraints cannot be dropped by using DROP INDEX. They are dropped using the ALTER TABLE DROP CONSTRAINT statement. For more information, see ALTER TABLE.

The following example deletes a clustered index with a PRIMARY KEY constraint by dropping the constraint. The `ProductCostHistory` table has no FOREIGN KEY constraints. If it did, those constraints would have to be removed first.

```
USE AdventureWorks2012;
GO
-- Set ONLINE = OFF to execute this example on editions other than Enterprise
Edition.
ALTER TABLE Production.TransactionHistoryArchive
DROP CONSTRAINT PK_TransactionHistoryArchive_TransactionID
WITH (ONLINE = ON);
```

```
GO
```

F. Dropping an XML index

The following example drops an XML index on the `ProductModel` table.

```
USE AdventureWorks2012;
GO
DROP INDEX PXML_ProductModel_CatalogDescription
    ON Production.ProductModel;
GO
```

G. Dropping a clustered index on a FILESTREAM table

The following example deletes a clustered index online and moves the resulting table (heap) and FILESTREAM data to the `MyPartitionScheme` partition scheme by using both the `MOVE TO` clause and the `FILESTREAM ON` clause.

```
USE MyDatabase;
GO
DROP INDEX PK_MyClusteredIndex
    ON dbo.MyTable
    MOVE TO MyPartitionScheme
    FILESTREAM_ON MyPartitionScheme;
GO
```

See Also

ALTER INDEX (Transact-SQL)
ALTER PARTITION SCHEME
ALTER TABLE
CREATE INDEX
CREATE PARTITION SCHEME
CREATE SPATIAL INDEX (Transact-SQL)
CREATE XML INDEX (Transact-SQL)
EVENTDATA (Transact-SQL)
sys.indexes
sys.tables
sys.filegroups
sp_spaceused

DROP LOGIN

Removes a SQL Server login account.

 Transact-SQL Syntax Conventions

Syntax

```
DROP LOGIN login_name
```

Arguments

login_name

Specifies the name of the login to be dropped.

Remarks

A login cannot be dropped while it is logged in. A login that owns any securable, server-level object, or SQL Server Agent job cannot be dropped.

You can drop a login to which database users are mapped; however, this will create orphaned users. For more information, see EVENTDATA (Transact-SQL).

Permissions

Requires ALTER ANY LOGIN permission on the server.

Examples

The following example drops the login `WilliJo`.

```
DROP LOGIN WilliJo;
GO
```

See Also

CREATE LOGIN (Transact-SQL)

ALTER LOGIN (Transact-SQL)

EVENTDATA (Transact-SQL)

DROP MASTER KEY

Removes the master key from the current database.

 Transact-SQL Syntax Conventions

Syntax

```
DROP MASTER KEY
```

Arguments

This statement takes no arguments.

Remarks

The drop will fail if any private key in the database is protected by the master key.

Permissions

Requires CONTROL permission on the database.

Examples

The following example removes the master key for the `AdventureWorks2012` database.

```
USE AdventureWorks2012;
DROP MASTER KEY;
GO
```

See Also

Encryption Hierarchy
OPEN MASTER KEY (Transact-SQL)
CLOSE MASTER KEY (Transact-SQL)
BACKUP MASTER KEY (Transact-SQL)
RESTORE MASTER KEY (Transact-SQL)
ALTER MASTER KEY (Transact-SQL)
Encryption Hierarchy

DROP MESSAGE TYPE

Drops an existing message type.

 Transact-SQL Syntax Conventions

Syntax

```
DROP MESSAGE TYPE message_type_name
[;]
```

Arguments

message_type_name

The name of the message type to delete. Server, database, and schema names cannot be specified.

Permissions

Permission for dropping a message type defaults to the owner of the message type, members of the db_ddladmin or db_owner fixed database roles, and members of the sysadmin fixed server role.

Remarks

You cannot drop a message type if any contracts refer to the message type.

Examples

The following example deletes the `//Adventure-Works.com/Expenses/SubmitExpense` message type from the database.

```
DROP MESSAGE TYPE [//Adventure-Works.com/Expenses/SubmitExpense] ;
```

See Also

EVENTDATA (Transact-SQL)

CREATE MESSAGE TYPE

EVENTDATA

DROP PARTITION FUNCTION

Removes a partition function from the current database. Partition functions are created by using CREATE PARTITION FUNCTION and modified by using ALTER PARTITION FUNCTION.

 Transact-SQL Syntax Conventions

Syntax

DROP PARTITION FUNCTION **partition_function_name** [;]

Arguments

partition_function_name

Is the name of the partition function that is to be dropped.

Remarks

A partition function can be dropped only if there are no partition schemes currently using the partition function. If there are partition schemes using the partition function, DROP PARTITION FUNCTION returns an error.

Permissions

Any one of the following permissions can be used to execute DROP PARTITION FUNCTION:

- ALTER ANY DATASPACE permission. This permission defaults to members of the **sysadmin** fixed server role and the **db_owner** and **db_ddladmin** fixed database roles.
- CONTROL or ALTER permission on the database in which the partition function was created.

- CONTROL SERVER or ALTER ANY DATABASE permission on the server of the database in which the partition function was created.

Examples

The following example assumes the partition function `myRangePF` has been created in the current database.

```
DROP PARTITION FUNCTION myRangePF;
```

See Also

sys.index_columns (Transact-SQL)
ALTER PARTITION FUNCTION
EVENTDATA
sys.partition_functions
sys.partition_parameters
sys.partition_range_values
sys.partitions
sys.tables
sys.indexes
sys.index_columns

DROP PARTITION SCHEME

Removes a partition scheme from the current database. Partition schemes are created by using CREATE PARTITION SCHEME and modified by using ALTER PARTITION SCHEME.

 Transact-SQL Syntax Conventions

Syntax

DROP PARTITION SCHEME **`partition_scheme_name`** [;]

Arguments

partition_scheme_name

Is the name of the partition scheme to be dropped.

Remarks

A partition scheme can be dropped only if there are no tables or indexes currently using the partition scheme. If there are tables or indexes using the partition scheme, DROP PARTITION SCHEME returns an error. DROP PARTITION SCHEME does not remove the filegroups themselves.

Permissions

The following permissions can be used to execute DROP PARTITION SCHEME:

- ALTER ANY DATASPACE permission. This permission defaults to members of the **sysadmin** fixed server role and the **db_owner** and **db_ddladmin** fixed database roles.
- CONTROL or ALTER permission on the database in which the partition scheme was created.
- CONTROL SERVER or ALTER ANY DATABASE permission on the server of the database in which the partition scheme was created.

Examples

The following example drops the partition scheme `myRangePS1` from the current database:

```
DROP PARTITION SCHEME myRangePS1;
```

See Also

sys.index_columns (Transact-SQL)
ALTER PARTITION SCHEME
sys.partition_schemes
EVENTDATA (Transact-SQL)
sys.data_spaces
sys.destination_data_spaces
sys.partitions
sys.tables
sys.indexes
sys.index_columns

DROP PROCEDURE

Removes one or more stored procedures or procedure groups from the current database in SQL Server 2012.

 Transact-SQL Syntax Conventions

Syntax

DROP { PROC | PROCEDURE } { [**schema_name.**] **procedure** } [**,...n**]

Arguments

schema_name

The name of the schema to which the procedure belongs. A server name or database name cannot be specified.

procedure

The name of the stored procedure or stored procedure group to be removed. Individual

procedures within a numbered procedure group cannot be dropped; the whole procedure group is dropped.

Best Practices

Before removing any stored procedure, check for dependent objects and modify these objects accordingly. Dropping a stored procedure can cause dependent objects and scripts to fail when these objects are not updated. For more information, see How to: View the dependencies of a stored procedure (SQL Server Management Studio)

Metadata

To display a list of existing procedures, query the **sys.objects** catalog view. To display the procedure definition, query the **sys.sql_modules** catalog view.

Security

Permissions

Requires **CONTROL** permission on the procedure, or **ALTER** permission on the schema to which the procedure belongs, or membership in the **db_ddladmin** fixed server role.

Examples

The following example removes the `dbo.uspMyProc` stored procedure in the current database.

```
DROP PROCEDURE dbo.uspMyProc;
GO
```

The following example removes several stored procedures in the current database.

```
DROP PROCEDURE dbo.uspGetSalesbyMonth, dbo.uspUpdateSalesQuotes,
dbo.uspGetSalesByYear;
```

See Also

ALTER PROCEDURE
CREATE PROCEDURE
sys.objects
sys.sql_modules
How to: Delete a stored procedure (SQL Server Management Studio)

DROP QUEUE

Drops an existing queue.

Transact-SQL Syntax Conventions

Syntax

```
DROP QUEUE <object>
[;]

<object> ::=
{
  [ database_name.[ schema_name ].| schema_name.]
    queue_name
}
```

Arguments

database_name

The name of the database that contains the queue to drop. When no database_name is provided, defaults to the current database.

schema_name (object)

The name of the schema that owns the queue to drop. When no schema_name is provided, defaults to the default schema for the current user.

queue_name

The name of the queue to drop.

Remarks

You cannot drop a queue if any services refer to the queue.

Permissions

Permission for dropping a queue defaults to the owner of the queue, members of the **db_ddladmin** or **db_owner** fixed database roles, and members of the **sysadmin** fixed server role.

Examples

The following example drops the **ExpenseQueue** queue from the current database.

```
DROP QUEUE ExpenseQueue ;
```

See Also

Queues
ALTER QUEUE
EVENTDATA

DROP REMOTE SERVICE BINDING

Drops a remote service binding.

 Transact-SQL Syntax Conventions

Syntax

```
DROP REMOTE SERVICE BINDING binding_name
[ ; ]
```

Arguments

binding_name

Is the name of the remote service binding to drop. Server, database, and schema names cannot be specified.

Permissions

Permission for dropping a remote service binding defaults to the owner of the remote service binding, members of the db_owner fixed database role, and members of the sysadmin fixed server role.

Examples

The following example deletes the remote service binding `APBinding` from the database.

```
DROP REMOTE SERVICE BINDING APBinding ;
```

See Also

EVENTDATA (Transact-SQL)

ALTER REMOTE SERVICE BINDING

EVENTDATA

DROP RESOURCE POOL

Drops a user-defined Resource Governor resource pool.

 Transact-SQL Syntax Conventions.

Syntax

```
DROP RESOURCE POOL pool_name
[ ; ]
```

Arguments

pool_name

Is the name of an existing user-defined resource pool.

Remarks

You cannot drop a resource pool if it contains workload groups.

You cannot drop the Resource Governor default or internal pools.

When you are executing DDL statements, we recommend that you be familiar with Resource Governor states. For more information, see Resource Governor.

Permissions

Requires CONTROL SERVER permission.

Examples

The following example drops the resource pool named `big_pool`.

```
DROP RESOURCE POOL big_pool
GO
ALTER RESOURCE GOVERNOR RECONFIGURE
GO
```

See Also

Resource Governor
CREATE RESOURCE POOL (Transact-SQL)
ALTER RESOURCE POOL (Transact-SQL)
CREATE WORKLOAD GROUP (Transact-SQL)
ALTER WORKLOAD GROUP (Transact-SQL)
DROP WORKLOAD GROUP (Transact-SQL)
ALTER RESOURCE GOVERNOR (Transact-SQL)

DROP ROLE

Removes a role from the database.

 Transact-SQL Syntax Conventions

Syntax

```
DROP ROLE role_name
```

Arguments

role_name

Specifies the role to be dropped from the database.

Remarks

Roles that own securables cannot be dropped from the database. To drop a database role that owns securables, you must first transfer ownership of those securables or drop them from the database. Roles that have members cannot be dropped from the database. To drop a role that has members, you must first remove members of the role.

To remove members from a database role, use ALTER ROLE (Transact-SQL).

You cannot use DROP ROLE to drop a fixed database role.

Information about role membership can be viewed in the sys.database_role_members catalog view.

Caution

Beginning with SQL Server 2005, the behavior of schemas changed. As a result, code that assumes that schemas are equivalent to database users may no longer return correct results. Old catalog views, including , should not be used in a database in which any of the following DDL statements have ever been used: CREATE SCHEMA, ALTER SCHEMA, DROP SCHEMA, CREATE USER, ALTER USER, DROP USER, CREATE ROLE, ALTER ROLE, DROP ROLE, CREATE APPROLE, ALTER APPROLE, DROP APPROLE, ALTER AUTHORIZATION. In such databases you must instead use the new catalog views. The new catalog views take into account the separation of principals and schemas that was introduced in SQL Server 2005. For more information about catalog views, see .

To remove a server role, use DROP SERVER ROLE (Transact-SQL).

Permissions

Requires **ALTER ANY ROLE** permission on the database, or **CONTOL** permission on the role, or membership in the **db_securityadmin**.

Examples

The following example drops the database role `purchasing` from `AdventureWorks2012`.

```
USE AdventureWorks2012;

DROP ROLE purchasing;

GO
```

See Also

CREATE ROLE (Transact-SQL)

ALTER ROLE (Transact-SQL)

Principals

eventdata (Transact-SQL)

sp_addrolemember (Transact-SQL)

sys.database_role_members (Transact-SQL)

sys.database_principals (Transact-SQL)

Security Functions (Transact-SQL)

DROP ROUTE

Drops a route, deleting the information for the route from the routing table of the current database.

Transact-SQL Syntax Conventions

Syntax

```
DROP ROUTE route_name
[;]
```

Arguments

route_name

The name of the route to drop. Server, database, and schema names cannot be specified.

Remarks

The routing table that stores the routes is a metadata table that can be read through the catalog view **sys.routes**. The routing table can only be updated through the CREATE ROUTE, ALTER ROUTE, and DROP ROUTE statements.

You can drop a route regardless of whether any conversations use the route. However, if there is no other route to the remote service, messages for those conversations will remain in the transmission queue until a route to the remote service is created or the conversation times out.

Permissions

Permission for dropping a route defaults to the owner of the route, members of the db_ddladmin or db_owner fixed database roles, and members of the sysadmin fixed server role.

Examples

The following example deletes the `ExpenseRoute` route.

```
DROP ROUTE ExpenseRoute ;
```

See Also

sys.routes (Transact-SQL)

CREATE ROUTE

EVENTDATA

sys.routes (Transact-SQL)

DROP RULE

Removes one or more user-defined rules from the current database.

Important
DROP RULE will be removed in the next version of Microsoft SQL Server. Do not use DROP RULE in new development work, and plan to modify applications that currently use them. Instead, use CHECK constraints that you can create by using the CHECK keyword of CREATE TABLE or ALTER TABLE. For more information, see Unique Constraints and Check Constraints.

Transact-SQL Syntax Conventions

Syntax

```
DROP RULE { [ schema_name . ] rule_name } [ ,...n ] [ ; ]
```

Arguments

schema_name

Is the name of the schema to which the rule belongs.

rule

Is the rule to be removed. Rule names must comply with the rules for identifiers. Specifying the rule schema name is optional.

Remarks

To drop a rule, first unbind it if the rule is currently bound to a column or to an alias data type. To unbind the rule, use **sp_unbindrule**. If the rule is bound when you try to drop it, an error message is displayed and the DROP RULE statement is canceled.

After a rule is dropped, new data entered into the columns previously governed by the rule is entered without the constraints of the rule. Existing data is not affected in any way.

The DROP RULE statement does not apply to CHECK constraints. For more information about dropping CHECK constraints, see ALTER TABLE (Transact-SQL).

Permissions

To execute DROP RULE, at a minimum, a user must have ALTER permission on the schema to which the rule belongs.

Examples

The following example unbinds and then drops the rule named `VendorID_rule`.

```
USE AdventureWorks;
GO
IF EXISTS (SELECT name FROM sysobjects
         WHERE name = 'VendorID_rule'
            AND type = 'R')
   BEGIN
```

```
        EXEC sp_unbindrule 'Production.ProductVendor.VendorID'
        DROP RULE VendorID_rule
    END
GO
```

See Also

CREATE RULE
sp_bindrule
sp_help
sp_helptext
sp_unbindrule
USE

DROP SCHEMA

Removes a schema from the database.

 Transact-SQL Syntax Conventions

Syntax

```
DROP SCHEMA schema_name
```

Arguments

schema_name

Is the name by which the schema is known within the database.

Remarks

The schema that is being dropped must not contain any objects. If the schema contains objects, the DROP statement fails.

Information about schemas is visible in the sys.schemas catalog view.

Caution Beginning with SQL Server 2005, the behavior of schemas changed. As a result, code that assumes that schemas are equivalent to database users may no longer return correct results. Old catalog views, including , should not be used in a database in which any of the following DDL statements have ever been used: CREATE SCHEMA, ALTER SCHEMA, DROP SCHEMA, CREATE USER, ALTER USER, DROP USER, CREATE ROLE, ALTER ROLE, DROP ROLE, CREATE APPROLE, ALTER APPROLE, DROP APPROLE, ALTER AUTHORIZATION. In such databases you must instead use the new catalog views. The new catalog views take into account the separation of principals and schemas that was introduced in SQL Server 2005. For more information about catalog views, see .

Permissions

Requires CONTROL permission on the schema or ALTER ANY SCHEMA permission on the database.

Examples

The following example starts with a single `CREATE SCHEMA` statement. The statement creates the schema `Sprockets` that is owned by `Krishna` and a table `Sprockets.NineProngs`, and then grants `SELECT` permission to `Anibal` and denies `SELECT` permission to `Hung-Fu`.

```
USE AdventureWorks2012;
GO
CREATE SCHEMA Sprockets AUTHORIZATION Krishna
    CREATE TABLE NineProngs (source int, cost int, partnumber int)
    GRANT SELECT TO Anibal
    DENY SELECT TO Hung-Fu;
GO
```

The following statements drop the schema. Note that you must first drop the table that is contained by the schema.

```
DROP TABLE Sprockets.NineProngs;
DROP SCHEMA Sprockets;
GO
```

See Also

CREATE SCHEMA (Transact-SQL)
ALTER SCHEMA (Transact-SQL)
DROP SCHEMA (Transact-SQL)
eventdata (Transact-SQL)

DROP SEARCH PROPERTY LIST

Drops a property list from the current database if the search property list is currently not associated with any full-text index in the database.

Important
CREATE SEARCH PROPERTY LIST, ALTER SEARCH PROPERTY LIST, and DROP SEARCH PROPERTY LIST are supported only under compatibility level 110. Under lower compatibility levels, these statements are not supported.

Syntax

```
DROP SEARCH PROPERTY LIST property_list_name
```

;

Arguments

property_list_name

Is the name of the search property list to be dropped. property_list_name is an identifier.

To view the names of the existing property lists, use the sys.registered_search_property_lists catalog view, as follows:

```
SELECT name FROM sys.registered_search_property_lists;
```

Remarks

You cannot drop a search property list from a database while the list is associated with any full-text index, and attempts to do so fail. To drop a search property list from a given full-text index, use the ALTER FULLTEXT INDEX statement, and specify the SET SEARCH PROPERTY LIST clause with either OFF or the name of another search property list.

To view the property lists on a server instance

- sys.registered_search_property_lists (Transact-SQL)

To view the property lists associated with full-text indexes

- sys.fulltext_indexes (Transact-SQL)

To remove a property list from a full-text index

- ALTER FULLTEXT INDEX (Transact-SQL)

Permissions

Requires CONTROL permission on the search property list.

Note
The property list owner can grant CONTROL permissions on the list. By default, the user who creates a search property list is its owner. The owner can be changed by using the ALTER AUTHORIZATION Transact-SQL statement.

Examples

The following example drops the `JobCandidateProperties` property list from the `AdventureWorks` database.

```
USE AdventureWorks;
GO
DROP SEARCH PROPERTY LIST JobCandidateProperties;
GO
```

See Also

ALTER SEARCH PROPERTY LIST (Transact-SQL)

CREATE SEARCH PROPERTY LIST (Transact-SQL)

Search Properties and Property Lists
sys.registered_search_properties (Transact-SQL)
sys.registered_search_property_lists (Transact-SQL)
sys.registered_search_property_lists (Transact-SQL)

DROP SEQUENCE

Removes a sequence object from the current database.

 Transact-SQL Syntax Conventions

Syntax

DROP SEQUENCE { [**database_name** . [**schema_name**] . | **schema_name**.] **sequence_name** } [,...n]
[;]

Arguments

database_name

Is the name of the database in which the sequence object was created.

schema_name

Is the name of the schema to which the sequence object belongs.

sequence_name

Is the name of the sequence to be dropped. Type is **sysname**.

Remarks

After generating a number, a sequence object has no continuing relationship to the number it generated, so the sequence object can be dropped, even though the number generated is still in use.

A sequence object can be dropped while it is referenced by a stored procedure, or trigger, because it is not schema bound. A sequence object cannot be dropped if it is referenced as a default value in a table. The error message will list the object referencing the sequence.

To list all sequence objects in the database, execute the following statement.

```
SELECT sch.name + '.' + seq.name AS [Sequence schema and name]
    FROM sys.sequences AS seq
    JOIN sys.schemas AS sch
        ON seq.schema_id = sch.schema_id ;
GO
```

Security

Permissions

Requires ALTER or CONTROL permission on the schema.

Audit

To audit **DROP SEQUENCE**, monitor the **SCHEMA_OBJECT_CHANGE_GROUP**.

Examples

The following example removes a sequence object named `CountBy1` from the current database.

```
DROP SEQUENCE CountBy1 ;
GO
```

See Also

ALTER SEQUENCE (Transact-SQL)

CREATE SEQUENCE (Transact-SQL)

NEXT VALUE FOR function (Transact-SQL)

Creating and Using Sequence Numbers

DROP SERVER AUDIT

Drops a Server Audit Object using the SQL Server Audit feature. For more information on SQL Server Audit, see Understanding SQL Server Audit.

 Transact-SQL Syntax Conventions

Syntax

```
DROP SERVER AUDIT audit_name
  [;]
```

Remarks

You must set the State of an audit to the OFF option in order to make any changes to an Audit. If DROP AUDIT is run while an audit is enabled with any options other than STATE=OFF, you will receive a MSG_NEED_AUDIT_DISABLED error message.

A DROP SERVER AUDIT removes the metadata for the Audit, but not the audit data that was collected before the command was issued.

DROP SERVER AUDIT does not drop associated server or database audit specifications. These specifications must be dropped manually or left orphaned and later mapped to a new server audit.

Permissions

To create, alter or drop a Server Audit Principals require the ALTER ANY SERVER AUDIT or the CONTROL SERVER permission.

Examples

The following example drops an audit called `HIPAA_Audit`.

```
ALTER SERVER AUDIT HIPAA_Audit
STATE = OFF;
GO
DROP SERVER AUDIT HIPAA_Audit;
GO
```

Updated content
Corrected the Permissions section.

See Also

CREATE SERVER AUDIT (Transact-SQL)
ALTER SERVER AUDIT (Transact-SQL)
CREATE SERVER AUDIT SPECIFICATION (Transact-SQL)
ALTER SERVER AUDIT SPECIFICATION (Transact-SQL)
DROP SERVER AUDIT SPECIFICATION (Transact-SQL)
CREATE DATABASE AUDIT SPECIFICATION (Transact-SQL)
ALTER DATABASE AUDIT SPECIFICATION (Transact-SQL)
DROP DATABASE AUDIT SPECIFICATION (Transact-SQL)
ALTER AUTHORIZATION (Transact-SQL)
fn_get_audit_file (Transact-SQL)
sys.server_audits (Transact-SQL)
sys.server_file_audits (Transact-SQL)
sys.server_audit_specifications (Transact-SQL)
sys.server_audit_specifications_details (Transact-SQL)
sys.database_audit_specifications (Transact-SQL)
sys.database_audit_specification_details (Transact-SQL)
sys.dm_server_audit_status
sys.dm_audit_actions
sys.dm_audit_class_type_map
Create a Server Audit and Server Audit Specification

DROP SERVER AUDIT SPECIFICATION

Drops a server audit specification object using the SQL Server Audit feature. For more information, see Understanding SQL Server Audit.

 Transact-SQL Syntax Conventions

Syntax

```
DROP SERVER AUDIT SPECIFICATION audit_specification_name
[ ; ]
```

Arguments

audit_specification_name

Name of an existing server audit specification object.

Remarks

A DROP SERVER AUDIT SPECIFICATION removes the metadata for the audit specification, but not the audit data collected before the DROP command was issued. You must set the state of a server audit specification to OFF using ALTER SERVER AUDIT SPECIFICATION before it can be dropped.

Permissions

Users with the ALTER ANY SERVER AUDIT permission can drop server audit specifications.

Examples

The following example drops a server audit specification called `HIPAA_Audit_Specification`.

```
DROP SERVER AUDIT SPECIFICATION HIPAA_Audit_Specification;
GO
```

For a full example about how to create an audit, see Understanding SQL Server Audit.

Updated content
Corrected the Permissions section.

See Also

CREATE SERVER AUDIT (Transact-SQL)
ALTER SERVER AUDIT (Transact-SQL)
DROP SERVER AUDIT (Transact-SQL)
CREATE SERVER AUDIT SPECIFICATION (Transact-SQL)
ALTER SERVER AUDIT SPECIFICATION (Transact-SQL)

CREATE DATABASE AUDIT SPECIFICATION (Transact-SQL)
ALTER DATABASE AUDIT SPECIFICATION (Transact-SQL)
DROP DATABASE AUDIT SPECIFICATION (Transact-SQL)
ALTER AUTHORIZATION (Transact-SQL)
fn_get_audit_file (Transact-SQL)
sys.server_audits (Transact-SQL)
sys.server_file_audits (Transact-SQL)
sys.server_audit_specifications (Transact-SQL)
sys.server_audit_specifications_details (Transact-SQL)
sys.database_audit_specifications (Transact-SQL)
sys.database_audit_specification_details (Transact-SQL)
sys.dm_server_audit_status
sys.dm_audit_actions
sys.dm_audit_class_type_map
Create a Server Audit and Server Audit Specification

DROP SERVER ROLE

Removes a user-defined server role.

User-defined server roles are new in SQL Server 2012.

 Transact-SQL Syntax Conventions

Syntax

```
DROP SERVER ROLE role_name
```

Arguments

role_name

Specifies the user-defined server role to be dropped from the server.

Remarks

User-defined server roles that own securables cannot be dropped from the server. To drop a user-defined server role that owns securables, you must first transfer ownership of those securables or delete them.

User-defined server roles that have members cannot be dropped. To drop a user-defined server role that has members, you must first remove members of the role by using ALTER SERVER ROLE.

Fixed server roles cannot be removed.

You can view information about role membership by querying the sys.server_role_members catalog view.

Permissions

Requires CONTROL permission on the server role or ALTER ANY SERVER ROLE permission.

Examples

A. To drop a server role

The following example drops the server role `purchasing`.

```
DROP SERVER ROLE purchasing;
GO
```

B. To view role membership

To view role membership, use the **Server Role (Members)** page in SQL Server Management Studio or execute the following query:

```
SELECT SRM.role_principal_id, SP.name AS Role_Name,
SRM.member_principal_id, SP2.name  AS Member_Name
FROM sys.server_role_members AS SRM
JOIN sys.server_principals AS SP
    ON SRM.Role_principal_id = SP.principal_id
JOIN sys.server_principals AS SP2
    ON SRM.member_principal_id = SP2.principal_id
ORDER BY  SP.name,  SP2.name
```

C. To view role membership

To determine whether a server role owns another server role, execute the following query:

```
SELECT SP1.name AS RoleOwner, SP2.name AS Server_Role
FROM sys.server_principals AS SP1
JOIN sys.server_principals AS SP2
    ON SP1.principal_id = SP2.owning_principal_id
ORDER BY SP1.name ;
```

See Also

ALTER ROLE (Transact-SQL)
CREATE ROLE (Transact-SQL)
Principals
DROP ROLE (Transact-SQL)
eventdata (Transact-SQL)

sp_addrolemember (Transact-SQL)
sys.database_role_members (Transact-SQL)
sys.database_principals (Transact-SQL)

DROP SERVICE

Drops an existing service.

 Transact-SQL Syntax Conventions

Syntax

```
DROP SERVICE service_name
[ ; ]
```

Arguments

service_name

The name of the service to drop. Server, database, and schema names cannot be specified.

Remarks

You cannot drop a service if any conversation priorities refer to it.

Dropping a service deletes all messages for the service from the queue that the service uses. Service Broker sends an error to the remote side of any open conversations that use the service.

Permissions

Permission for dropping a service defaults to the owner of the service, members of the db_ddladmin or db_owner fixed database roles, and members of the sysadmin fixed server role.

Examples

The following example drops the service `//Adventure-Works.com/Expenses`.

```
DROP SERVICE [//Adventure-Works.com/Expenses] ;
```

See Also

ALTER BROKER PRIORITY (Transact-SQL)
ALTER SERVICE (Transact-SQL)
CREATE SERVICE (Transact-SQL)
DROP BROKER PRIORITY (Transact-SQL)
EVENTDATA (Transact-SQL)

DROP SIGNATURE

Drops a digital signature from a stored procedure, function, trigger, or assembly.

Transact-SQL Syntax Conventions

Syntax

```
DROP [ COUNTER ] SIGNATURE FROM module_name
    BY <crypto_list> [ ,...n ]

<crypto_list> ::=
    CERTIFICATE cert_name
    | ASYMMETRIC KEY Asym_key_name
```

Arguments

module_name

Is the name of a stored procedure, function, assembly, or trigger.

CERTIFICATE cert_name

Is the name of a certificate with which the stored procedure, function, assembly, or trigger is signed.

ASYMMETRIC KEY Asym_key_name

Is the name of an asymmetric key with which the stored procedure, function, assembly, or trigger is signed.

Remarks

Information about signatures is visible in the sys.crypt_properties catalog view.

Permissions

Requires ALTER permission on the object and CONTROL permission on the certificate or asymmetric key. If an associated private key is protected by a password, the user also must have the password.

Examples

The following example removes the signature of certificate `HumanResourcesDP` from the stored procedure `HumanResources.uspUpdateEmployeeLogin`.

```
USE AdventureWorks2012;
DROP SIGNATURE FROM HumanResources.uspUpdateEmployeeLogin
    BY CERTIFICATE HumanResourcesDP;
GO
```

See Also

sys.crypt_properties (Transact-SQL)

ADD SIGNATURE (Transact-SQL)

DROP STATISTICS

Drops statistics for multiple collections within the specified tables in the current database.

 Transact-SQL Syntax Conventions

Syntax

DROP STATISTICS **table.statistics_name** | **view.statistics_name** [**,...n**]

Arguments

table | view

Is the name of the target table or indexed view for which statistics should be dropped. Table and view names must comply with the rules for identifiers. Specifying the table or view owner name is optional.

statistics_name

Is the name of the statistics group to drop. Statistics names must comply with the rules for identifiers

Remarks

Be careful when you drop statistics. Doing so may affect the execution plan chosen by the query optimizer.

Statistics on indexes cannot be dropped by using DROP STATISTICS. Statistics remain as long as the index exists.

For more information about displaying statistics, see DBCC SHOW_STATISTICS (Transact-SQL).

Permissions

Requires ALTER permission on the table or view.

Examples

The following example drops the statistics groups (collections) of two tables. The `VendorCredit` statistics group (collection) of the `Vendor` table and the `CustomerTotal` statistics (collection) of the `SalesOrderHeader` table are dropped.

```
-- Create the statistics groups.
USE AdventureWorks2012;
GO
CREATE STATISTICS VendorCredit
    ON Purchasing.Vendor (Name, CreditRating)
    WITH SAMPLE 50 PERCENT
CREATE STATISTICS CustomerTotal
    ON Sales.SalesOrderHeader (CustomerID, TotalDue)
```

```
    WITH FULLSCAN;
GO
DROP STATISTICS Purchasing.Vendor.VendorCredit,
Sales.SalesOrderHeader.CustomerTotal;
```

See Also

ALTER DATABASE
CREATE INDEX
CREATE STATISTICS
sys.stats
sys.stats_columns
DBCC SHOW_STATISTICS
sp_autostats
sp_createstats
UPDATE STATISTICS
EVENTDATA
USE

DROP SYMMETRIC KEY

Removes a symmetric key from the current database.

 Transact-SQL Syntax Conventions

Syntax

DROP SYMMETRIC KEY **`symmetric_key_name`** [REMOVE PROVIDER KEY]

Arguments

symmetric_key_name

Is the name of the symmetric key to be dropped.

REMOVE PROVIDER KEY

Removes an Extensible Key Management (EKM) key from an EKM device. For more information about Extensible Key Management, see Understanding Extensible Key Management (EKM).

Remarks

If the key is open in the current session the statement will fail.

If the asymmetric key is mapped to an Extensible Key Management (EKM) key on an EKM device and the **REMOVE PROVIDER KEY** option is not specified, the key will be dropped from the database but not the device, and a warning will be issued.

Permissions

Requires CONTROL permission on the symmetric key.

Examples

The following example removes a symmetric key named `GailSammamishKey6` from the current database.

```
CLOSE SYMMETRIC KEY GailSammamishKey6;
DROP SYMMETRIC KEY GailSammamishKey6;
GO
```

See Also

CREATE SYMMETRIC KEY (Transact-SQL)
CLOSE SYMMETRIC KEY (Transact-SQL)
ALTER SYMMETRIC KEY (Transact-SQL)
Encryption Hierarchy
CLOSE SYMMETRIC KEY (Transact-SQL)
Understanding Extensible Key Management (EKM)

DROP SYNONYM

Removes a synonym from a specified schema.

 Transact-SQL Syntax Conventions

Syntax

```
DROP SYNONYM [ schema. ] synonym_name
```

Arguments

schema

Specifies the schema in which the synonym exists. If schema is not specified, SQL Server uses the default schema of the current user.

synonym_name

Is the name of the synonym to be dropped.

Remarks

References to synonyms are not schema-bound; therefore, you can drop a synonym at any time. References to dropped synonyms will be found only at run time.

Synonyms can be created, dropped and referenced in dynamic SQL.

Permissions

To drop a synonym, a user must satisfy at least one of the following conditions. The user must be:

- The current owner of a synonym.
- A grantee holding CONTROL on a synonym.
- A grantee holding ALTER SCHEMA permission on the containing schema.

Examples

The following example first creates a synonym, `MyProduct`, and then drops the synonym.

```
USE tempdb;
GO
-- Create a synonym for the Product table in AdventureWorks2012.
CREATE SYNONYM MyProduct
FOR AdventureWorks2012.Production.Product;
GO
-- Drop synonym MyProduct.
USE tempdb;
GO
DROP SYNONYM MyProduct;
GO
```

See Also

CREATE SYNONYM

EVENTDATA

DROP TABLE

Removes one or more table definitions and all data, indexes, triggers, constraints, and permission specifications for those tables. Any view or stored procedure that references the dropped table must be explicitly dropped by using DROP VIEW or DROP PROCEDURE. To report the dependencies on a table, use sys.dm_sql_referencing_entities.

Syntax

```
DROP TABLE [ database_name . [ schema_name ] . | schema_name . ]
    table_name [ ,...n ] [ ; ]
```

Arguments

database_name

Is the name of the database in which the table was created.

schema_name

Is the name of the schema to which the table belongs.

table_name

Is the name of the table to be removed.

Remarks

DROP TABLE cannot be used to drop a table that is referenced by a FOREIGN KEY constraint. The referencing FOREIGN KEY constraint or the referencing table must first be dropped. If both the referencing table and the table that holds the primary key are being dropped in the same DROP TABLE statement, the referencing table must be listed first.

Multiple tables can be dropped in any database. If a table being dropped references the primary key of another table that is also being dropped, the referencing table with the foreign key must be listed before the table holding the primary key that is being referenced.

When a table is dropped, rules or defaults on the table lose their binding, and any constraints or triggers associated with the table are automatically dropped. If you re-create a table, you must rebind the appropriate rules and defaults, re-create any triggers, and add all required constraints.

If you delete all rows in a table by using DELETE tablename or use the TRUNCATE TABLE statement, the table exists until it is dropped.

Large tables and indexes that use more than 128 extents are dropped in two separate phases: logical and physical. In the logical phase, the existing allocation units used by the table are marked for deallocation and locked until the transaction commits. In the physical phase, the IAM pages marked for deallocation are physically dropped in batches.

If you drop a table that contains a VARBINARY(MAX) column with the FILESTREAM attribute, any data stored in the file system will not be removed.

Important
DROP TABLE and CREATE TABLE should not be executed on the same table in the same batch. Otherwise an unexpected error may occur.

Permissions

Requires ALTER permission on the schema to which the table belongs, CONTROL permission on the table, or membership in the **db_ddladmin** fixed database role.

Examples

A. Dropping a table in the current database

The following example removes the `ProductVendor1` table and its data and indexes from the current database.

```
DROP TABLE ProductVendor1 ;
```

B. Dropping a table in another database

The following example drops the `SalesPerson2` table in the database. The example can be executed from any database on the server instance.

```
DROP TABLE AdventureWorks2012.dbo.SalesPerson2 ;
```

C. Dropping a temporary table

The following example creates a temporary table, tests for its existence, drops it, and tests again for its existence.

```
USE AdventureWorks2012;
GO
CREATE TABLE #temptable (col1 int);
GO
INSERT INTO #temptable
VALUES (10);
GO
SELECT * FROM #temptable;
GO
IF OBJECT_ID(N'tempdb..#temptable', N'U') IS NOT NULL
DROP TABLE #temptable;
GO
--Test the drop.
SELECT * FROM #temptable;
```

See Also

ALTER TABLE
CREATE TABLE
DELETE (Transact-SQL)
sp_help (Transact-SQL)
sp_spaceused (Transact-SQL)
TRUNCATE TABLE (Transact-SQL)
DROP VIEW (Transact-SQL)
DROP PROCEDURE (Transact-SQL)
EVENTDATA (Transact-SQL)
sys.sql_expression_dependencies (Transact-SQL)

DROP TRIGGER

Removes one or more DML or DDL triggers from the current database.

 Transact-SQL Syntax Conventions

Syntax

Trigger on an INSERT, UPDATE, or DELETE statement to a table or view (DML Trigger)

DROP TRIGGER [**schema_name**.]**trigger_name** [**,**...**n**] [;]

Trigger on a CREATE, ALTER, DROP, GRANT, DENY, REVOKE or UPDATE statement (DDL Trigger)

DROP TRIGGER **trigger_name** [**,**...**n**]
ON { DATABASE | ALL SERVER }
[;]

Trigger on a LOGON event (Logon Trigger)

DROP TRIGGER **trigger_name** [**,**...**n**]
ON ALL SERVER

Arguments

schema_name

Is the name of the schema to which a DML trigger belongs. DML triggers are scoped to the schema of the table or view on which they are created. schema_name cannot be specified for DDL or logon triggers.

trigger_name

Is the name of the trigger to remove. To see a list of currently created triggers, use sys.server_assembly_modules or sys.server_triggers.

DATABASE

Indicates the scope of the DDL trigger applies to the current database. DATABASE must be specified if it was also specified when the trigger was created or modified.

ALL SERVER

Indicates the scope of the DDL trigger applies to the current server. ALL SERVER must be specified if it was also specified when the trigger was created or modified. ALL SERVER also applies to logon triggers.

Note

This option is not available in a contained database.

Remarks

You can remove a DML trigger by dropping it or by dropping the trigger table. When a table is dropped, all associated triggers are also dropped.

When a trigger is dropped, information about the trigger is removed from the **sys.objects**, **sys.triggers** and **sys.sql_modules** catalog views.

Multiple DDL triggers can be dropped per DROP TRIGGER statement only if all triggers were created using identical ON clauses.

To rename a trigger, use DROP TRIGGER and CREATE TRIGGER. To change the definition of a trigger, use ALTER TRIGGER.

For more information about determining dependencies for a specific trigger, see sys.sql_expression_dependencies, sys.dm_sql_referenced_entities, and sys.dm_sql_referencing_entities.

For more information about viewing the text of the trigger, see sp_helptext and sys.sql_modules.

For more information about viewing a list of existing triggers, see sys.triggers and sys.server_triggers.

Permissions

To drop a DML trigger requires ALTER permission on the table or view on which the trigger is defined.

To drop a DDL trigger defined with server scope (ON ALL SERVER) or a logon trigger requires CONTROL SERVER permission in the server. To drop a DDL trigger defined with database scope (ON DATABASE) requires ALTER ANY DATABASE DDL TRIGGER permission in the current database.

Examples

A. Dropping a DML trigger

The following example drops the `employee_insupd` trigger.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID ('employee_insupd', 'TR') IS NOT NULL
   DROP TRIGGER employee_insupd;
GO
```

B. Dropping a DDL trigger

The following example drops DDL trigger `safety`.

Important

Because DDL triggers are not schema-scoped and, therefore do not appear in the **sys.objects** catalog view, the OBJECT_ID function cannot be used to query whether they exist in the database. Objects that are not schema-scoped must be queried by using the appropriate catalog view. For DDL triggers, use **sys.triggers**.

```
USE AdventureWorks2012;
GO
IF EXISTS (SELECT * FROM sys.triggers
    WHERE parent_class = 0 AND name = 'safety')
DROP TRIGGER safety
ON DATABASE;
GO
```

See Also

ALTER TRIGGER (Transact-SQL)
CREATE TRIGGER (Transact-SQL)
ENABLE TRIGGER (Transact-SQL)
DISABLE TRIGGER (Transact-SQL)
eventdata (Transact-SQL)
Getting Information About DML Triggers
sp_help (Transact-SQL)
sp_helptrigger (Transact-SQL)
sys.triggers (Transact-SQL)
sys.trigger_events (Transact-SQL)
sys.sql_modules (Transact-SQL)
sys.assembly_modules (Transact-SQL)
sys.server_triggers (Transact-SQL)
sys.server_trigger_events (Transact-SQL)
sys.server_sql_modules (Transact-SQL)
sys.server_assembly_modules (Transact-SQL)

DROP TYPE

Removes an alias data type or a common language runtime (CLR) user-defined type from the current database.

 Transact-SQL Syntax Conventions

Syntax

DROP TYPE [**schema_name.**] **type_name** [;]

Arguments

schema_name

Is the name of the schema to which the alias or user-defined type belongs.

type_name

Is the name of the alias data type or the user-defined type you want to drop.

Remarks

The DROP TYPE statement will not execute when any of the following is true:

- There are tables in the database that contain columns of the alias data type or the user-defined type. Information about alias or user-defined type columns can be obtained by querying the sys.columns or sys.column_type_usages catalog views.
- There are computed columns, CHECK constraints, schema-bound views, and schema-bound functions whose definitions reference the alias or user-defined type. Information about these references can be obtained by querying the sys.sql_expression_dependencies catalog view.
- There are functions, stored procedures, or triggers created in the database, and these routines use variables and parameters of the alias or user-defined type. Information about alias or user-defined type parameters can be obtained by querying the sys.parameters or sys.parameter_type_usages catalog views.

Permissions

Requires either CONTROL permission on type_name or ALTER permission on schema_name.

Examples

The following example assumes a type named `ssn` is already created in the current database.

```
DROP TYPE ssn ;
```

See Also

CREATE TYPE (Transact-SQL)

EVENTDATA (Transact-SQL)

DROP USER

Removes a user from the current database.

Transact-SQL Syntax Conventions

Syntax

DROP USER **user_name**

Arguments

user_name

Specifies the name by which the user is identified inside this database.

Remarks

Users that own securables cannot be dropped from the database. Before dropping a database user that owns securables, you must first drop or transfer ownership of those securables.

The guest user cannot be dropped, but guest user can be disabled by revoking its CONNECT permission by executing REVOKE CONNECT FROM GUEST within any database other than master or tempdb.

Caution

Beginning with SQL Server 2005, the behavior of schemas changed. As a result, code that assumes that schemas are equivalent to database users may no longer return correct results. Old catalog views, including , should not be used in a database in which any of the following DDL statements have ever been used: CREATE SCHEMA, ALTER SCHEMA, DROP SCHEMA, CREATE USER, ALTER USER, DROP USER, CREATE ROLE, ALTER ROLE, DROP ROLE, CREATE APPROLE, ALTER APPROLE, DROP APPROLE, ALTER AUTHORIZATION. In such databases you must instead use the new catalog views. The new catalog views take into account the separation of principals and schemas that was introduced in SQL Server 2005. For more information about catalog views, see .

Permissions

Requires ALTER ANY USER permission on the database.

Examples

The following example removes database user `AbolrousHazem` from the `AdventureWorks2012` database.

```
USE AdventureWorks2012;

DROP USER AbolrousHazem;

GO
```

See Also

CREATE USER (Transact-SQL)

ALTER USER (Transact-SQL)

eventdata (Transact-SQL)

DROP VIEW

Removes one or more views from the current database. DROP VIEW can be executed against indexed views.

Transact-SQL Syntax Conventions

Syntax

DROP VIEW [**schema_name** .] **view_name** [...,**n**] [;]

Arguments

schema_name

Is the name of the schema to which the view belongs.

view_name

Is the name of the view to remove.

Remarks

When you drop a view, the definition of the view and other information about the view is deleted from the system catalog. All permissions for the view are also deleted.

Any view on a table that is dropped by using DROP TABLE must be dropped explicitly by using DROP VIEW.

When executed against an indexed view, DROP VIEW automatically drops all indexes on a view. To display all indexes on a view, use sp_helpindex.

When querying through a view, the Database Engine checks to make sure that all the database objects referenced in the statement exist and that they are valid in the context of the statement, and that data modification statements do not violate any data integrity rules. A check that fails returns an error message. A successful check translates the action into an action against the underlying table or tables. If the underlying tables or views have changed since the view was originally created, it may be useful to drop and re-create the view.

For more information about determining dependencies for a specific view, see USE (Transact-SQL).

For more information about viewing the text of the view, see sp_helptext.

Permissions

Requires **CONTROL** permission on the view, **ALTER** permission on the schema containing the view, or membership in the **db_ddladmin** fixed server role.

Examples

The following example removes the view `Reorder`.

```
USE AdventureWorks2012 ;
GO
IF OBJECT_ID ('dbo.Reorder', 'V') IS NOT NULL
DROP VIEW dbo.Reorder ;
GO
```

See Also

ALTER VIEW
CREATE VIEW
EVENTDATA
sys.columns
sys.objects
USE
sys.sql_expression_dependencies (Transact-SQL)

DROP WORKLOAD GROUP

Drops an existing user-defined Resource Governor workload group.

 Transact-SQL Syntax Conventions.

Syntax

```
DROP WORKLOAD GROUP group_name
[;]
```

Arguments

group_name

Is the name of an existing user-defined workload group.

Remarks

The DROP WORKLOAD GROUP statement is not allowed on the Resource Governor internal or default groups.

When you are executing DDL statements, we recommend that you be familiar with Resource Governor states. For more information, see Resource Governor.

If a workload group contains active sessions, dropping or moving the workload group to a different resource pool will fail when the ALTER RESOURCE GOVERNOR RECONFIGURE statement is called to apply the change. To avoid this problem, you can take one of the following actions:

- Wait until all the sessions from the affected group have disconnected, and then rerun the ALTER RESOURCE GOVERNOR RECONFIGURE statement.
- Explicitly stop sessions in the affected group by using the KILL command, and then rerun the ALTER RESOURCE GOVERNOR RECONFIGURE statement.
- Restart the server. After the restart process is completed, the deleted group will not be created, and a moved group will use the new resource pool assignment.

- In a scenario in which you have issued the DROP WORKLOAD GROUP statement but decide that you do not want to explicitly stop sessions to apply the change, you can re-create the group by using the same name that it had before you issued the DROP statement, and then move the group to the original resource pool. To apply the changes, run the ALTER RESOURCE GOVERNOR RECONFIGURE statement.

Permissions

Requires CONTROL SERVER permission.

Examples

The following example drops the workload group named `adhoc`.

```
DROP WORKLOAD GROUP adhoc
GO
ALTER RESOURCE GOVERNOR RECONFIGURE
GO
```

See Also

Resource Governor
CREATE WORKLOAD GROUP (Transact-SQL)
ALTER WORKLOAD GROUP (Transact-SQL)
CREATE RESOURCE POOL (Transact-SQL)
ALTER RESOURCE POOL (Transact-SQL)
DROP RESOURCE POOL (Transact-SQL)
ALTER RESOURCE GOVERNOR (Transact-SQL)

DROP XML SCHEMA COLLECTION

Deletes the whole XML schema collection and all of its components.

 Transact-SQL Syntax Conventions

Syntax

DROP XML SCHEMA COLLECTION [**relational_schema.**]**sql_identifier**

Arguments

relational_schema

Identifies the relational schema name. If not specified, the default relational schema is assumed.

sql_identifier

Is the name of the XML schema collection to drop.

Remarks

Dropping an XML schema collection is a transactional operation. This means when you drop an XML schema collection inside a transaction and later roll back the transaction, the XML schema collection is not dropped.

You cannot drop an XML schema collection when it is in use. This means that the collection being dropped cannot be any of the following:

- Associated with any **xml** type parameter or column.
- Specified in any table constraints.
- Referenced in a schema-bound function or stored procedure. For example, the following function will lock the XML schema collection `MyCollection` because the function specifies `WITH SCHEMABINDING`. If you remove it, there is no lock on the XML SCHEMA COLLECTION.

```
CREATE FUNCTION dbo.MyFunction()
RETURNS int
WITH SCHEMABINDING
AS
BEGIN
   ...
   DECLARE @x XML(MyCollection)
   ...
END
```

Permissions

To drop an XML SCHEMA COLLECTION requires DROP permission on the collection.

Examples

The following example shows removing an XML schema collection.

```
DROP XML SCHEMA COLLECTION ManuInstructionsSchemaCollection
GO
```

See Also

CREATE XML SCHEMA COLLECTION (Transact-SQL)

ALTER XML SCHEMA COLLECTION (Transact-SQL)

EVENTDATA (Transact-SQL)

Typed vs. Untyped XML

Guidelines and Limitations of XML Schemas on the Server

ENABLE TRIGGER

Enables a DML, DDL, or logon trigger.

 Transact-SQL Syntax Conventions

Syntax

```
ENABLE TRIGGER { [ schema_name . ] trigger_name [ ,...n ] | ALL }
ON { object_name | DATABASE | ALL SERVER } [ ; ]
```

Arguments

schema_name

Is the name of the schema to which the trigger belongs. schema_name cannot be specified for DDL or logon triggers.

trigger_name

Is the name of the trigger to be enabled.

ALL

Indicates that all triggers defined at the scope of the ON clause are enabled.

object_name

Is the name of the table or view on which the DML trigger trigger_name was created to execute.

DATABASE

For a DDL trigger, indicates that trigger_name was created or modified to execute with database scope.

ALL SERVER

For a DDL trigger, indicates that trigger_name was created or modified to execute with server scope. ALL SERVER also applies to logon triggers.

 **Note**

This option is not available in a contained database.

Remarks

Enabling a trigger does not re-create it. A disabled trigger still exists as an object in the current database, but does not fire. Enabling a trigger causes it to fire when any Transact-SQL statements on which it was originally programmed are executed. Triggers are disabled by using DISABLE TRIGGER. DML triggers defined on tables can be also be disabled or enabled by using ALTER TABLE.

Permissions

To enable a DML trigger, at a minimum, a user must have ALTER permission on the table or view on which the trigger was created.

To enable a DDL trigger with server scope (ON ALL SERVER) or a logon trigger, a user must have CONTROL SERVER permission on the server. To enable a DDL trigger with database scope (ON DATABASE), at a minimum, a user must have ALTER ANY DATABASE DDL TRIGGER permission in the current database.

Examples

A. Enabling a DML trigger on a table

The following example disables trigger `uAddress` that was created on table `Address`, and then enables it.

```
USE AdventureWorks2012;
GO
DISABLE TRIGGER Person.uAddress ON Person.Address;
GO
ENABLE Trigger Person.uAddress ON Person.Address;
GO
```

B. Enabling a DDL trigger

The following example creates a DDL trigger `safety` with database scope, and then disables it.

```
IF EXISTS (SELECT * FROM sys.triggers
    WHERE parent_class = 0 AND name = 'safety')
DROP TRIGGER safety ON DATABASE;
GO
CREATE TRIGGER safety
ON DATABASE
FOR DROP_TABLE, ALTER_TABLE
AS
   PRINT 'You must disable Trigger "safety" to drop or alter tables!'
   ROLLBACK;
GO
DISABLE TRIGGER safety ON DATABASE;
GO
ENABLE TRIGGER safety ON DATABASE;
GO
```

C. Enabling all triggers that were defined with the same scope

The following example enables all DDL triggers that were created at the server scope.

```
USE AdventureWorks2012;
GO
ENABLE Trigger ALL ON ALL SERVER;
GO
```

See Also

sys.triggers (Transact-SQL)
ALTER TRIGGER
CREATE TRIGGER
DROP TRIGGER
sys.triggers

UPDATE STATISTICS

Updates query optimization statistics on a table or indexed view. By default, the query optimizer already updates statistics as necessary to improve the query plan; in some cases you can improve query performance by using UPDATE STATISTICS or the stored procedure sp_updatestats to update statistics more frequently than the default updates.

Updating statistics ensures that queries compile with up-to-date statistics. However, updating statistics causes queries to recompile. We recommend not updating statistics too frequently because there is a performance tradeoff between improving query plans and the time it takes to recompile queries. The specific tradeoffs depend on your application. UPDATE STATISTICS can use tempdb to sort the sample of rows for building statistics.

 Transact-SQL Syntax Conventions

Syntax

```
UPDATE STATISTICS table_or_indexed_view_name
  [
    {
      { index_or_statistics__name }
     | ( { index_or_statistics_name } [ ,...n ] )
        }
  ]
  [   WITH
```

```
    [
       FULLSCAN
       | SAMPLE number { PERCENT | ROWS }
       | RESAMPLE
       | <update_stats_stream_option> [ ,...n ]
    ]
    [ [ , ] [ ALL | COLUMNS | INDEX ]
    [ [ , ] NORECOMPUTE ]
] ;
```

<update_stats_stream_option> ::=

```
[ STATS_STREAM = stats_stream ]
[ ROWCOUNT = numeric_constant ]
[ PAGECOUNT = numeric_contant ]
```

Arguments

table_or_indexed_view_name

Is the name of the table or indexed view to update statistics on.

index_or_statistics_name

Is the name of the index to update statistics on or name of the statistics to update. If index_or_statistics_name is not specified, the query optimizer updates all statistics for the table or indexed view. This includes statistics created using the CREATE STATISTICS statement, single-column statistics created when AUTO_CREATE_STATISTICS is on, and statistics created for indexes.

For more information about AUTO_CREATE_STATISTICS, see ALTER DATABASE SET Options (Transact-SQL). To view all indexes for a table or view, you can use sp_helpindex.

FULLSCAN

Compute statistics by scanning all rows in the table or indexed view. FULLSCAN and SAMPLE 100 PERCENT have the same results. FULLSCAN cannot be used with the SAMPLE option.

SAMPLE number { PERCENT | ROWS }

Specifies the approximate percentage or number of rows in the table or indexed view for the query optimizer to use when it updates statistics. For PERCENT, number can be from 0 through 100 and for ROWS, number can be from 0 to the total number of rows. The actual percentage or number of rows the query optimizer samples might not match the percentage or number specified. For example, the query optimizer scans all rows on a data page.

SAMPLE is useful for special cases in which the query plan, based on default sampling, is not

optimal. In most situations, it is not necessary to specify SAMPLE because the query optimizer uses sampling and determines the statistically significant sample size by default, as required to create high-quality query plans.

SAMPLE cannot be used with the FULLSCAN option. When neither SAMPLE nor FULLSCAN is specified, the query optimizer uses sampled data and computes the sample size by default.

We recommend against specifying 0 PERCENT or 0 ROWS. When 0 PERCENT or ROWS is specified, the statistics object is updated but does not contain statistics data.

RESAMPLE

Update each statistic using its most recent sample rate.

Using RESAMPLE can result in a full-table scan. For example, statistics for indexes use a full-table scan for their sample rate. When none of the sample options (SAMPLE, FULLSCAN, RESAMPLE) are specified, the query optimizer samples the data and computes the sample size by default.

ALL | COLUMNS | INDEX

Update all existing statistics, statistics created on one or more columns, or statistics created for indexes. If none of the options are specified, the UPDATE STATISTICS statement updates all statistics on the table or indexed view.

NORECOMPUTE

Disable the automatic statistics update option, AUTO_UPDATE_STATISTICS, for the specified statistics. If this option is specified, the query optimizer completes this statistics update and disables future updates.

To re-enable the AUTO_UPDATE_STATISTICS option behavior, run UPDATE STATISTICS again without the NORECOMPUTE option or run **sp_autostats**.

Warning

Using this option can produce suboptimal query plans. We recommend using this option sparingly, and then only by a qualified system administrator.

For more information about the AUTO_STATISTICS_UPDATE option, see ALTER DATABASE SET Options (Transact-SQL).

<update_stats_stream_option>

Identified for informational purposes only. Not supported. Future compatibility is not guaranteed.

Remarks

When to Use UPDATE STATISTICS

For more information about when to use UPDATE STATISTICS, see Statistics.

Updating All Statistics with sp_updatestats

For information about how to update statistics for all user-defined and internal tables in the database, see the stored procedure sp_updatestats (Transact-SQL). For example, the following command calls sp_updatestats to update all statistics for the database.

```
EXEC sp_updatestats;
```

Determining the Last Statistics Update

To determine when statistics were last updated, use the STATS_DATE function.

Permissions

Requires ALTER permission on the table or view.

Examples

A. Update all statistics on a table

The following example updates the statistics for all indexes on the `SalesOrderDetail` table.

```
USE AdventureWorks2012;
GO
UPDATE STATISTICS Sales.SalesOrderDetail;
```

GO B. Update the statistics for an index

The following example updates the statistics for the `AK_SalesOrderDetail_rowguid` index of the `SalesOrderDetail` table.

```
USE AdventureWorks2012;
GO
UPDATE STATISTICS Sales.SalesOrderDetail AK_SalesOrderDetail_rowguid;
GO
```

C. Update statistics by using 50 percent sampling

The following example creates and then updates the statistics for the `Name` and `ProductNumber` columns in the `Product` table.

```
USE AdventureWorks2012;
GO
CREATE STATISTICS Products
    ON Production.Product ([Name], ProductNumber)
    WITH SAMPLE 50 PERCENT
-- Time passes. The UPDATE STATISTICS statement is then executed.
UPDATE STATISTICS Production.Product(Products)
```

```
    WITH SAMPLE 50 PERCENT;
```

D. Update statistics by using FULLSCAN and NORECOMPUTE

The following example updates the `Products` statistics in the `Product` table, forces a full scan of all rows in the `Product` table, and turns off automatic statistics for the `Products` statistics.

```
USE AdventureWorks2012;
GO
UPDATE STATISTICS Production.Product(Products)
    WITH FULLSCAN, NORECOMPUTE;
GO
```

See Also

Statistics
ALTER DATABASE
CREATE STATISTICS
DBCC SHOW_STATISTICS
DROP STATISTICS
sp_autostats
sp_updatestats
STATS_DATE

TRUNCATE TABLE

Removes all rows from a table without logging the individual row deletions. TRUNCATE TABLE is similar to the DELETE statement with no WHERE clause; however, TRUNCATE TABLE is faster and uses fewer system and transaction log resources.

 Transact-SQL Syntax Conventions

Syntax

```
TRUNCATE TABLE
   [ { database_name .[ schema_name ] . | schema_name . } ]
   table_name
[ ; ]
```

Arguments

database_name

Is the name of the database.

schema_name

Is the name of the schema to which the table belongs.

table_name

Is the name of the table to truncate or from which all rows are removed.

Remarks

Compared to the DELETE statement, TRUNCATE TABLE has the following advantages:

- Less transaction log space is used.

  The DELETE statement removes rows one at a time and records an entry in the transaction log for each deleted row. TRUNCATE TABLE removes the data by deallocating the data pages used to store the table data and records only the page deallocations in the transaction log.
- Fewer locks are typically used.

  When the DELETE statement is executed using a row lock, each row in the table is locked for deletion. TRUNCATE TABLE always locks the table and page but not each row.
- Without exception, zero pages are left in the table.

  After a DELETE statement is executed, the table can still contain empty pages. For example, empty pages in a heap cannot be deallocated without at least an exclusive (LCK_M_X) table lock. If the delete operation does not use a table lock, the table (heap) will contain many empty pages. For indexes, the delete operation can leave empty pages behind, although these pages will be deallocated quickly by a background cleanup process.

TRUNCATE TABLE removes all rows from a table, but the table structure and its columns, constraints, indexes, and so on remain. To remove the table definition in addition to its data, use the DROP TABLE statement.

If the table contains an identity column, the counter for that column is reset to the seed value defined for the column. If no seed was defined, the default value 1 is used. To retain the identity counter, use DELETE instead.

Restrictions

You cannot use TRUNCATE TABLE on tables that:

- Are referenced by a FOREIGN KEY constraint. (You can truncate a table that has a foreign key that references itself.)
- Participate in an indexed view.
- Are published by using transactional replication or merge replication.

For tables with one or more of these characteristics, use the DELETE statement instead.

TRUNCATE TABLE cannot activate a trigger because the operation does not log individual row deletions. For more information, see CREATE TRIGGER (Transact-SQL).

Truncating Large Tables

Microsoft SQL Server has the ability to drop or truncate tables that have more than 128 extents without holding simultaneous locks on all the extents required for the drop.

Permissions

The minimum permission required is ALTER on table_name. TRUNCATE TABLE permissions default to the table owner, members of the sysadmin fixed server role, and the db_owner and db_ddladmin fixed database roles, and are not transferable. However, you can incorporate the TRUNCATE TABLE statement within a module, such as a stored procedure, and grant appropriate permissions to the module using the EXECUTE AS clause.

Examples

The following example removes all data from the `JobCandidate` table. `SELECT` statements are included before and after the `TRUNCATE TABLE` statement to compare results.

```
USE AdventureWorks2012;
GO
SELECT COUNT(*) AS BeforeTruncateCount
FROM HumanResources.JobCandidate;
GO
TRUNCATE TABLE HumanResources.JobCandidate;
GO
SELECT COUNT(*) AS AfterTruncateCount
FROM HumanResources.JobCandidate;
GO
```

See Also

DELETE (Transact-SQL)
DROP TABLE (Transact-SQL)
IDENTITY (Property) (Transact-SQL)