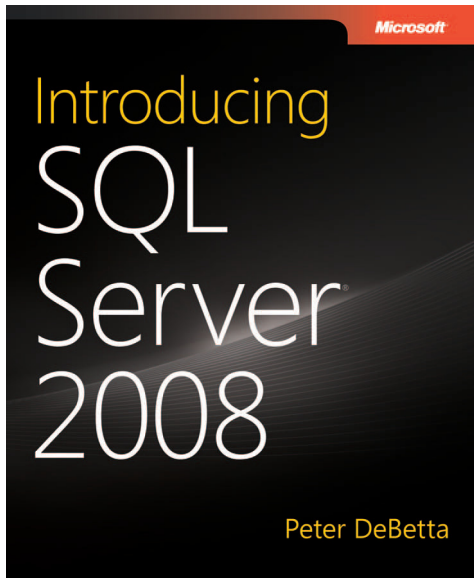


Introducing  
SQL  
Server®  
2008

Peter DeBetta



# Introducing SQL Server<sup>®</sup> 2008

*Peter DeBetta*

**PREVIEW CONTENT** This excerpt contains uncorrected manuscript from an upcoming Microsoft Press title, for early preview, and is subject to change prior to release. This excerpt is from *Introducing SQL Server<sup>®</sup> 2008* from Microsoft Press (ISBN 978-0-7356-2558-7, copyright 2008 Microsoft Corporation, all rights reserved), and is provided without any express, statutory, or implied warranties

To learn more about this book, visit Microsoft Learning at  
<http://www.microsoft.com/MSPress/books/12316.aspx>

978-0-7356-2558-7

**Microsoft<sup>®</sup>**  
*Press*

© 2008 Microsoft Corporation. All rights reserved.

## Chapter 1

# Declarative Management Framework

Have you ever had to ensure that only Windows logons or groups were added to Microsoft SQL Server, or that `xp_cmdshell` was disabled, or that no stored procedures were named starting with "sp\_"? Did you ever have to do this to more than one server in your enterprise? I have, and it was always such a hassle to go from server instance to server instance, querying system objects, checking various configuration settings, and scouring through all sorts of places to ensure that your SQL Server instances were all compliant. That process has changed in SQL Server 2008.

## Compliance in SQL Server 2008

Yes, it's true. SQL Server 2008 introduces a new feature known as the Declarative Management Framework, or more fondly, the DMF. The DMF allows you to define policies on a variety of objects and then either manually or automatically enforce said policies. Management is also very simple using SQL Server Management Studio (preferred), or you can write your own code to manage policies. But I am getting ahead of myself. Let's start at the beginning....

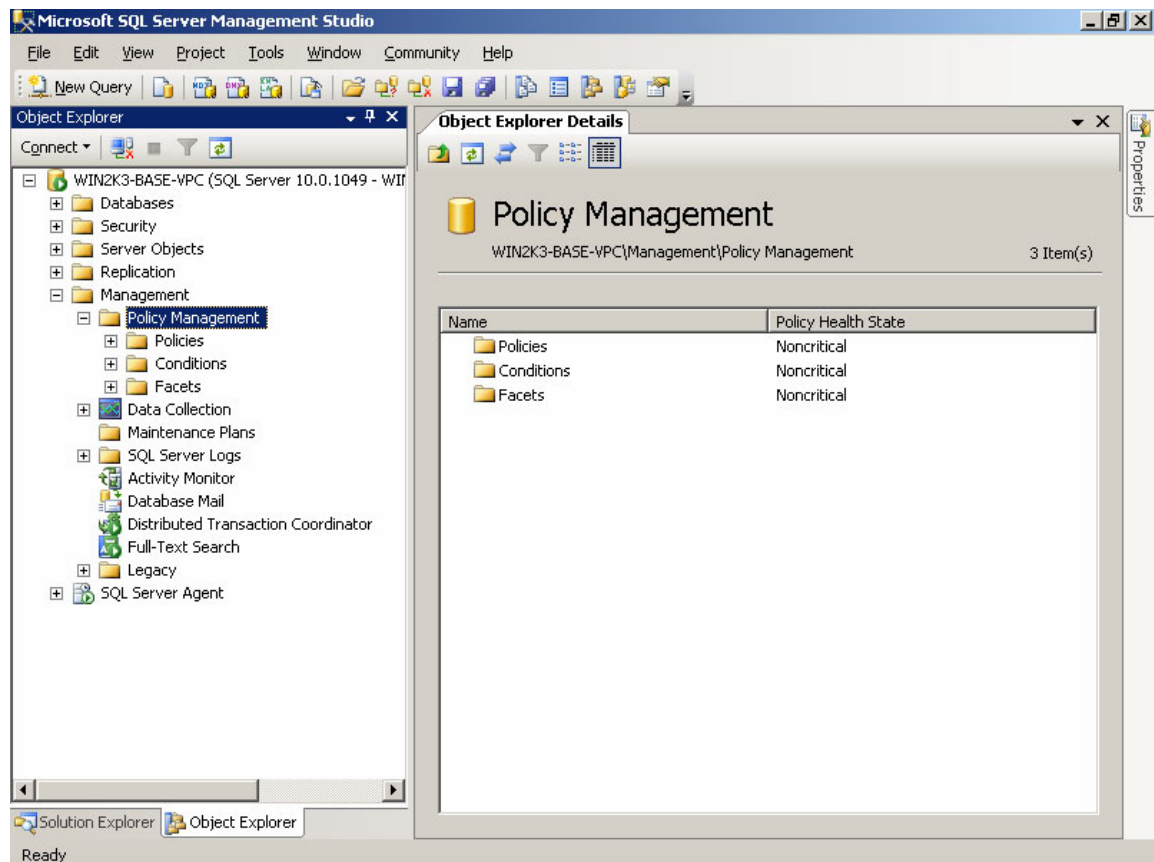
The DMF allows you to easily and proactively manage a variety of policies, ranging from security to metadata. It does this in two ways:

- The DMF allows you to monitor changes to these policies, with options to manually check policies, log violations, or even possibly automatically enforce them.
- The DMF allows you to manage one or more SQL Server instances, on a single server, or across multiple servers.

Rather than waiting for something to go awry, you can set up policies based on your server specifications and then have the DMF proactively enforce these policies or inform you via policy logs when these policies are out of compliance. The ability to enforce depends on the type of feature, or *facet* for which you are creating a policy. For example, if you want ensure that `xp_cmdshell` is never turned on for any sever you are managing, you can create a policy and have it inform you when a change occurs, or even have it check for changes on a schedule, but you cannot enforce it. The ability to enforce varies from facet to facet.

## The DMF in SQL Server Management Studio

The practice of creating and enforcing policies is easily achieved using SQL Server Management Studio. The DMF is accessed primarily by the Policy Management node in Object Explorer, which can be found under the Management node of the SQL Server instance, as shown in Figure 1-1.



**Figure 1-1** Policy Management in Object Explorer.

Within this node of Object Explorer, you find the three base items of the DMF: Policies, Conditions, and Facets. Although not shown as a node, *Policy Category Management* can also be accessed from here by right-clicking on the *Policy Management* node of Object Explorer and choosing *Manage Categories*. So what does each of the objects do to help you manage compliance? Let's dig into each of them in more detail and discover how they are used.

## DMF Objects

The DMF actually uses five different objects which are used to manage policies: Facets, Conditions, Policies, Target Sets, and Categories.

### Facets

Facets are the base units of the DMF. Facets are types of objects, such as a Surface Area feature, server, logon, database, user, and so on. Each facet has a set of predefined properties against which conditions can be created.

There are a total of 38 facets, with a whopping 1330 total properties. SQL Server Management Studio has a list of these facets under the Facets node in Objects Explorer (found under Management, Policy Management). Alas, if you want to see each list of properties, you need to open each facet's properties individually. If you want a quick list of

all facets and properties, however, you can use the SQL Server Management Objects (SMO) to iterate through all available facets and properties, as shown here:

```
FacetInfoCollection fic = PolicyStore.Facets;
IEnumerable<FacetInfo> fic_sorted =  from fic_i in fic
                                   orderby fic_i.DisplayName
                                   select fic_i;

Int32 pcount;
foreach (FacetInfo fi in fic_sorted)
{
    Console.WriteLine("FACET: " + fi.DisplayName);
    IEnumerable<PropertyInfo> fi_sorted =  from fi_i in fi.FacetProperties
                                           orderby fi_i.Name
                                           select fi_i;

    pcount = 0;
    foreach (PropertyInfo pi in fi_sorted)
    {
        if (pcount++ > 0)
            Console.Write(", " + pi.Name);
        else
            Console.Write(pi.Name);
    }
    Console.WriteLine();
    Console.ReadLine();
}
Console.WriteLine("---End of List---");
Console.ReadLine();
```

Listing 1

Facets by themselves cannot do anything in establishing compliance policies. They can be used by conditions, however, to define what compliance rules you want to create and against which servers, databases, or other objects the compliance should be checked.

## Conditions

A condition is a group of one or more facet properties, and their respective state. You express a condition by setting a facet property, a comparative operator, and a value. Each property condition's state is set according to its respective data type. For example, the Name property of the Stored Procedure facet is of type String, and can have a condition operator of equal (=), not equal (!=), LIKE, NOT LIKE, IN, or NOT IN, and thus can be compared to a string or a list of strings. The SQL Mail property of the Surface Area facet is of data type Boolean, and thus has only the equality and inequality operators, and can only be set to a value of true or false.

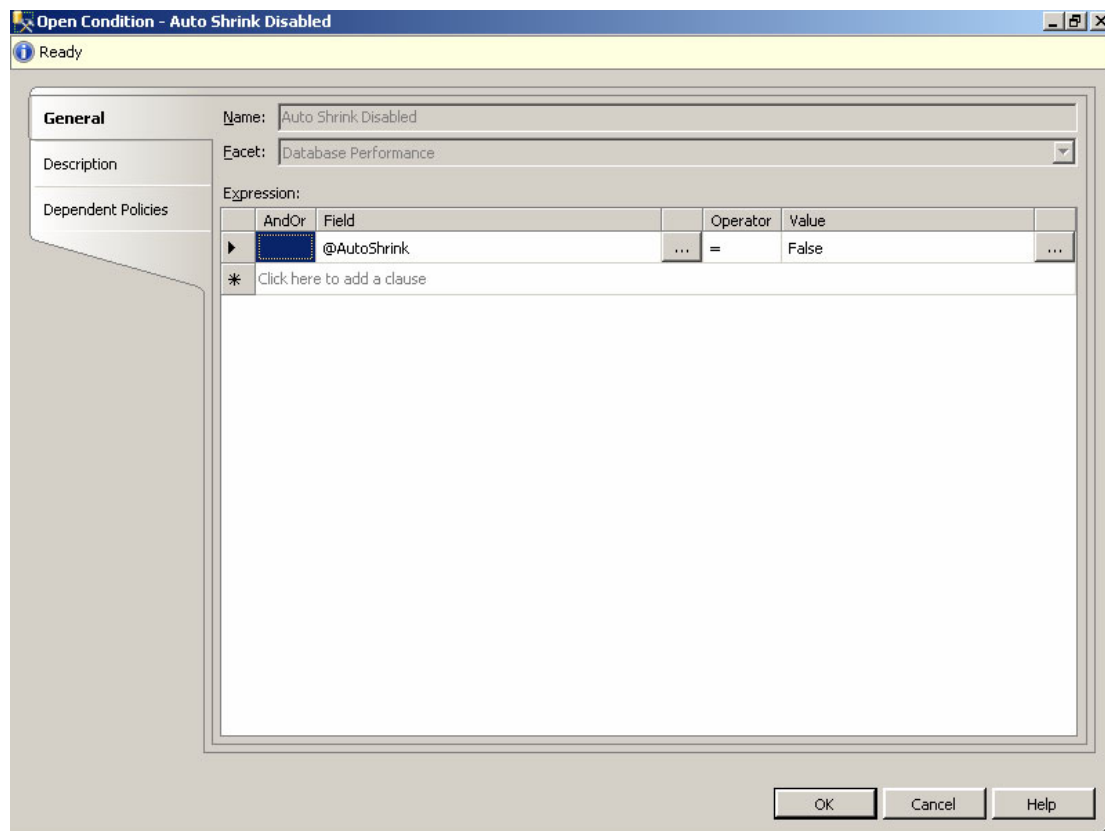
**Note** There is an advanced expression editor (The Advanced Edit dialog box) available if you need to create a specialized condition check. For example, you can check that the name of a table doesn't equal the schema name, or that all tables have a primary key. The advanced expression editor allows a lot of flexibility, but when used in a condition, its respective policy can only be executed *On Demand*.

Both the field and expression value can be set using the advanced expression editor, and in addition to providing a custom expression, it also provides an explanation of the available functions and a description of the facet properties, so if you are not sure what the property represents, you do not need to go to the facet and open it, you can simply click the Ellipses (...) button and examine the properties from there.

Furthermore, a condition can also only contain properties from a single facet type. For example, you can create a condition that states “SQL Mail is disabled and Database Mail is disabled” since both of these properties are part of the Surface Area Facet. You cannot, however, create a condition that states “stored procedure names must begin with ‘pr’ and xp\_cmdshell is disabled” since these two properties are part of two different facets (Stored Procedure Facet and Surface Area Facet, respectively).

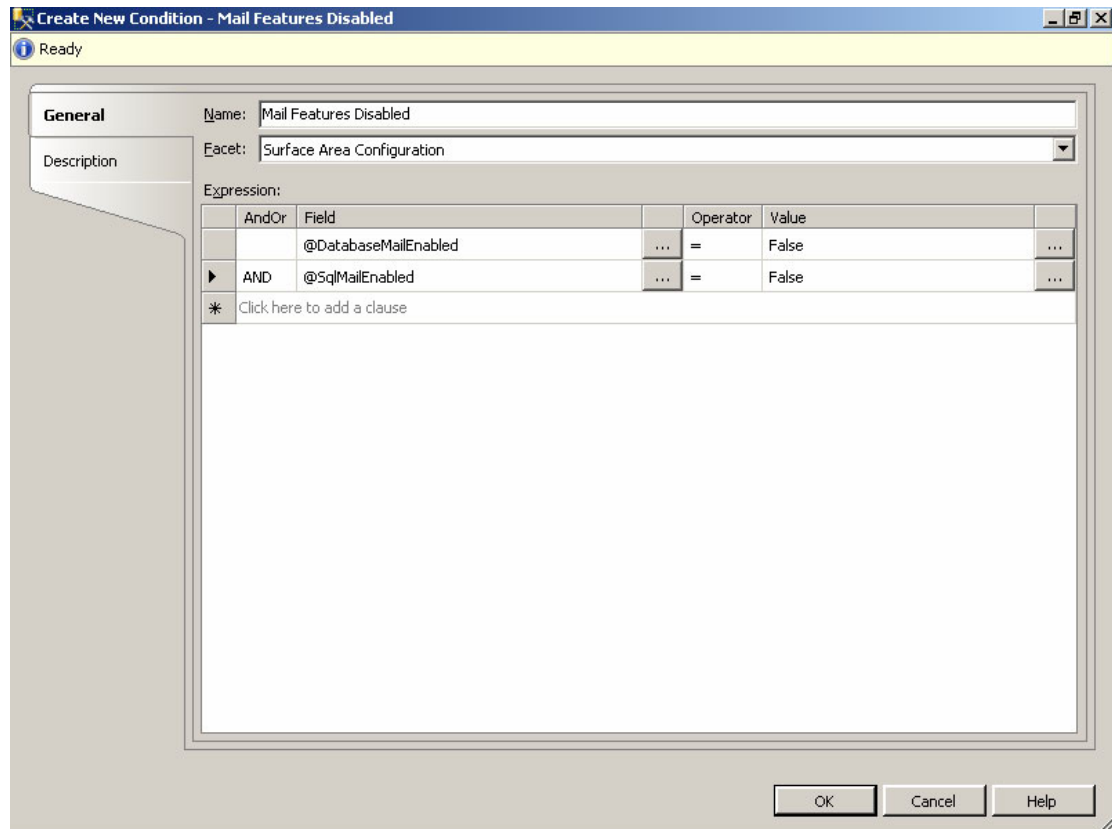
You can, however, create multiple conditions based on the same underlying facets. So you can create a condition that states “SQL Mail is disabled and Database Mail is disabled” and create a second condition that states “SQL Mail is disabled and Database Mail is enabled”. Of course, you wouldn’t want to check compliance for both of these on the same server, since you will always be out of compliance for one of them.

SQL Server 2008 comes with an assortment of predefined conditions that you can immediately put into use. For example, one of my favorites is the Condition named “Auto Shrink Disabled”, which can be used by a policy to ensure that databases do not enable the auto shrink option. Figure 1-2 shows this particular condition in the Open Condition window.



**Figure 1-2** The Open Condition Window.

As I stated earlier in this section, you can also set multiple property states in a condition. Multiple conditions can each be set with an *OR* or *AND* clause, and follows the standard order of operations. For example, Figure 1-3 shows an example of a new condition named “Mail Features Disabled” that states both SQL Mail and Database Mail are disabled.



**Figure 1-3** A New Condition for Disabled Mail Features.

So now you can define one or more conditions that you want to enforce on schedule, but how does a condition actually get enforced? The answer starts with the use of a policy.

## Policies

A policy is associated to a single condition and can be set to enforce or check the condition on one or more servers. The Execution Mode of the policy determines how a policy is enforced. Execution Mode can be set to one of four values:

- **On demand.** Do not check or enforce the policy—used to manually check policies.
- **On schedule, log out-of-compliance.** Check the policy on a set schedule and log if out of compliance.
- **Changes are attempted, log out-of-compliance.** Check the policy whenever a change occurs to the associated facet properties and log if out of compliance.
- **Changes are attempted, prevent out-of-compliance.** Check the policy whenever a change occurs to the associated facet properties and enforce compliance.

All policies can have an execution mode of *On demand* or *On schedule, log out-of-compliance*. Only some, however, can be set to *Changes are attempted, log out-of-compliance* or *Changes are attempted, prevent out-of-compliance*. The execution mode setting of a policy is determined by the policy's condition's underlying facet. Properties of certain facets can be enforced to prevent out-of-compliance when a change is attempted,

whereas other facets can be checked on changes but only log out-of-compliance, and still others only checked on schedule.

**Note** The execution mode also determines whether or not the policy needs to be enabled or not. If the execution mode is set to On Demand, then the policy must be disabled. For all other execution modes, the policy can be enabled or disabled as needed. Keep in mind that if a policy is disabled, even if its execution mode is set to *Changes are attempted, prevent out-of-compliance*, it will not be checked, and will not be automatically enforced.

How can you tell which facets support which execution modes? A quick query of the `syspolicy_management_facets` system view can give you the answer:

```

; WITH AutomatedPolicyExecutionMode (ModeId, ModeName)
AS
(
  SELECT *
  FROM (VALUES (0, 'On demand')
             , (1, 'Enforce Compliance')
             , (2, 'Check on change and log')
             , (4, 'Check on schedule and log')) AS EM(ModeId, ModeName)
)
SELECT
  pmf.[management_facet_id] AS FacetID
  , pmf.[name] AS FacetName
  , APEM.[ModeName]
FROM syspolicy_management_facets AS pmf
  INNER JOIN AutomatedPolicyExecutionMode AS APEM
  ON pmf.[execution_mode] & APEM.[ModeId] = APEM.[ModeId]
ORDER BY pmf.[name], APEM.[ModeName]

```

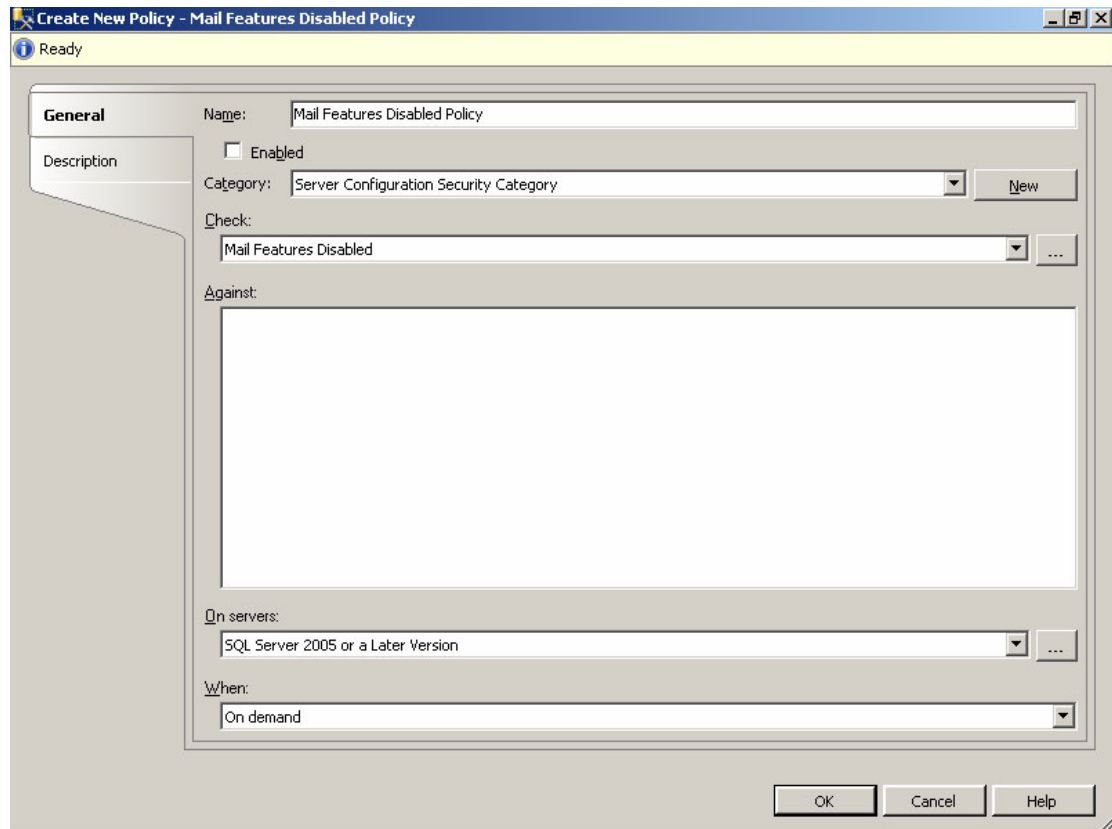
Listing 2

This query will show you a list of facets and their supported execution modes. Abridged results are shown here:

FacetID	FacetName	ModeName
1	ApplicationRole	On demand
1	ApplicationRole	Enforce Compliance
1	ApplicationRole	Check on change and log
1	ApplicationRole	Check on schedule and log
2	AsymmetricKey	On demand
2	AsymmetricKey	Check on schedule and log
3	BackupDevice	On demand
3	BackupDevice	Check on schedule and log
4	Database	On demand
4	Database	Check on change and log
4	Database	Check on schedule and log

Listing 3

How the policy is enforced is only a part of the concept of policies. Figure 1-4 shows an example of a policy that checks the Mail Features Disabled condition, but also does it only in SQL Server 2005 or later.

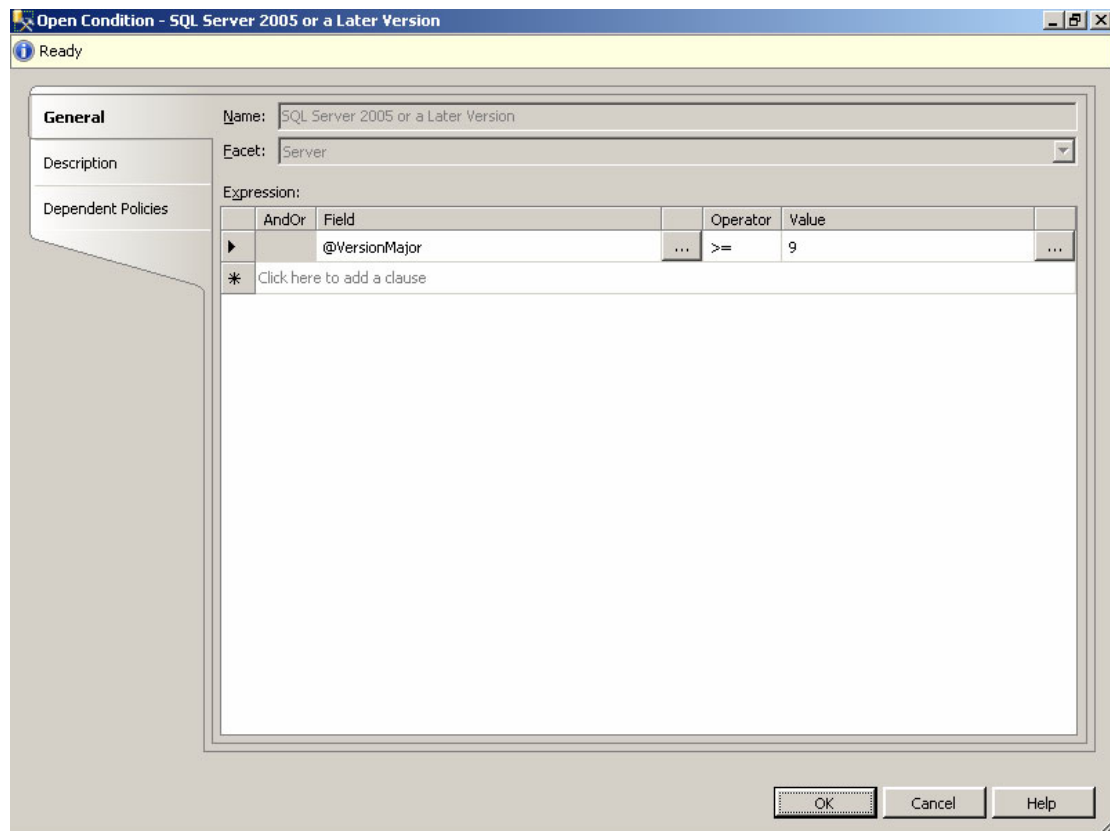


**Figure 1-4** Policy.

## Target Sets

Conditions are the basis for compliance in policies, but they can also be used to filter policies against target sets. A target set consists of one or more objects in the object hierarchy (i.e., server, database, and table) and conditions used to filter which of these objects the policy checks. Target sets are broken into two categories. The first is for the server, which is used to filter those servers the policy checks for compliance. The second is for the hierarchy of the database and its child objects (tables, views, stored procedures, and so on). For example, suppose you are implementing a policy for a condition that states Database Mail and SQL Mail are disabled. Such a policy wouldn't be applicable for SQL Server 2000, so you would want the policy to apply only to SQL Server 2005 or a later version.

The first step would be to create a condition for Database Mail and SQL Mail disabled (as shown in a previous example for the condition named "Mail Features Disabled"). Next you would create a condition for the server's major version greater than or equal to 9 (SQL Server 2000 is version 8, 2005 is version 9, and 2008 is version 10), as shown in Figure 1-5. This condition named "SQL Server 2005 or a Later Version" is actually created as a predefined condition on installation of SQL Server 2008.



**Figure 1-5** Condition for SQL Server version 9 (2005) or greater.

Finally, you would create the Policy that would check the “Mail Features Disabled” condition, but additionally specify the “SQL Server 2005 or a Later Version” condition in the *On Servers* drop-down list, as shown earlier in Figure 1-4.

Another way to use a condition to define a target set is by way of the *Against* section of the policy, which allows you to create the target set for the database hierarchy. For example, perhaps you want to enforce a naming convention for tables such that tables cannot start with the prefix “tbl”. First you create a condition named “Table Name” on the *Table* facet that states: @Name NOT LIKE 'tbl%'. From here, you create a policy named “Table Name Best Practice” that checks the “Table Name” condition. Next, in the *Against* section, you specify that the check is only done against “Non System Tables” in “Online User Database” (two more predefined conditions that come installed with SQL Server 2008). Now the check would only apply to non-system tables in online user databases. Figure 1-6 shows this policy and how you can choose the target condition for database.

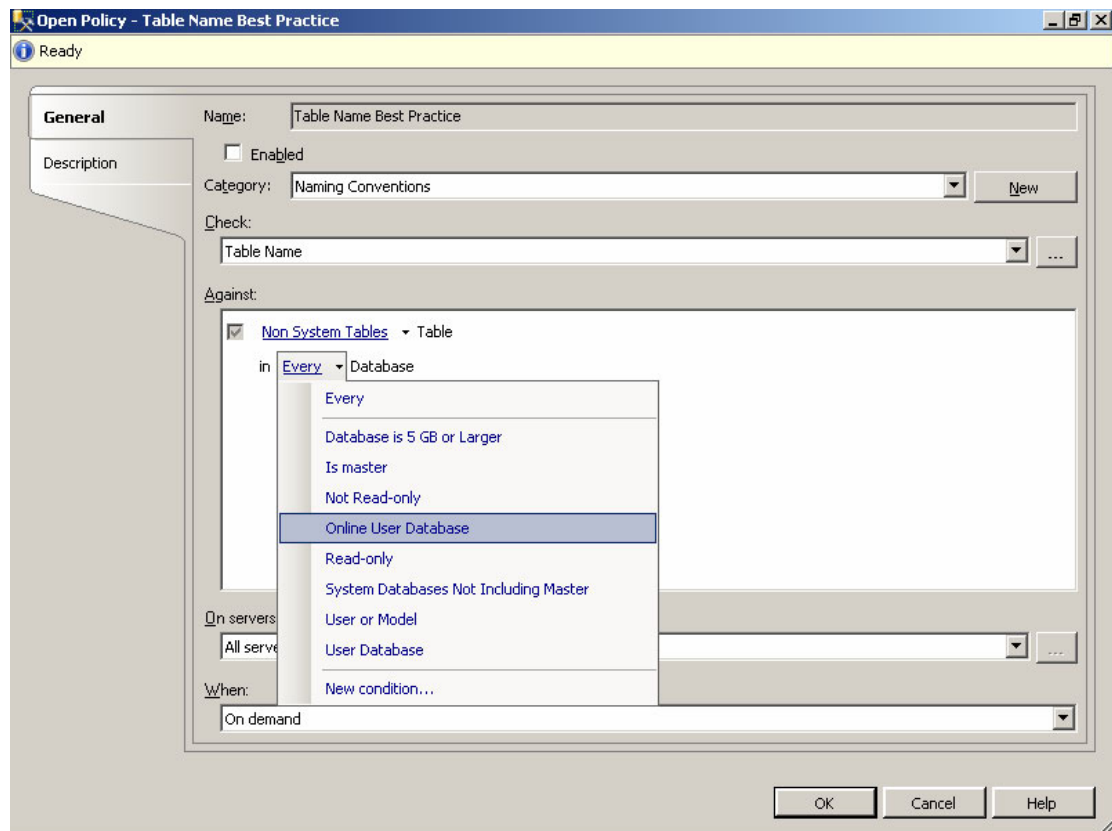


Figure 1-6

### Policy with Server filter.

You may have noticed that the policy that used the “Mail Features Disabled” condition didn’t have any option available in the *Against* section. The reason for this is that the *Against* section applies only to objects lower than server in the hierarchy. The *On Servers* option applies for servers themselves, and so if your policy is based on a condition that is at the server level (Server, Server Performance, Server Configuration, and so on), it will not have any options for lower level target sets (such as databases, tables, columns, and so on).

By using targets, however, in conjunction with condition checks, you can use different types of underlying facets in a single policy by using a condition for the check, and for each level in the object hierarchy.

### Policy Categories

Policy categories are used to group policies, and can be used to force compliance, or allow database owners to optionally subscribe to a set of policies. A policy can belong to only one policy category, either user-defined, or the *Default* category. Policy categories can be created on the fly when defining a policy, and can be further managed through the *Manage Policy Categories* dialog box. In this dialog box, one can also determine if category subscriptions at the database level are mandated or optional, as shown in Figure 1-7.

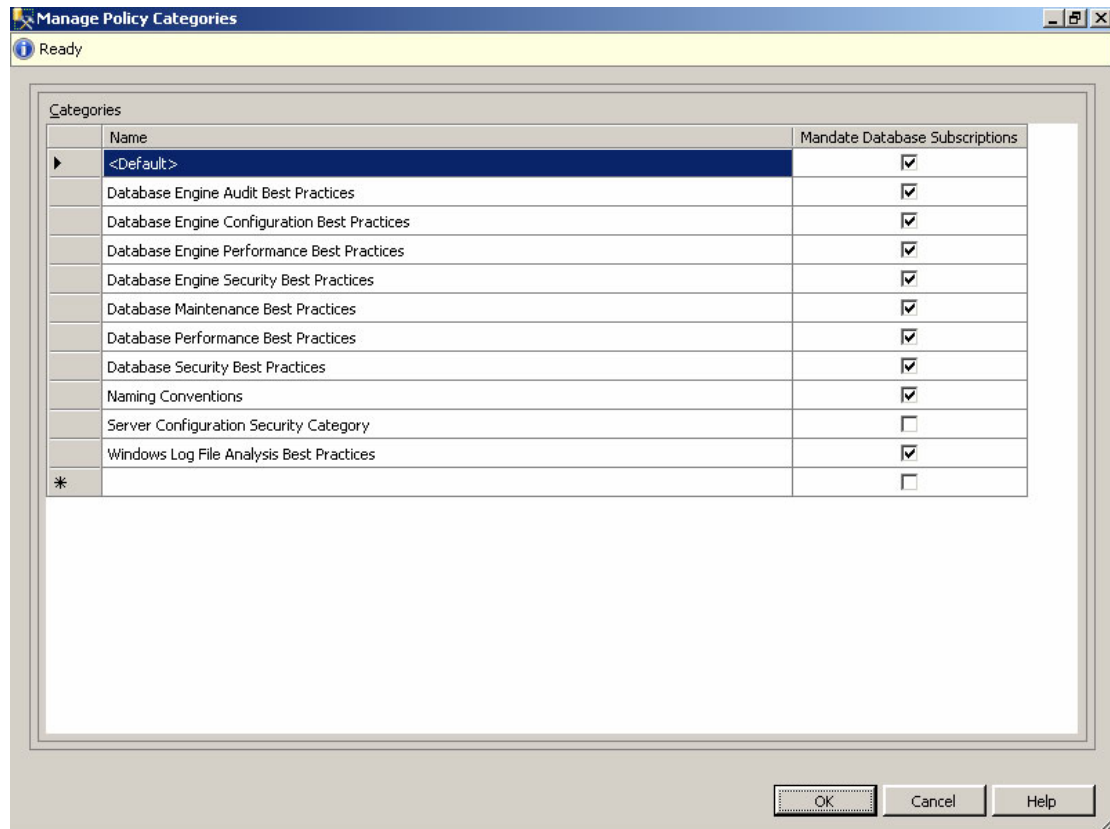


Figure 1-7

### The Manage Policy Categories Dialog Box.

Yes, you read that correctly—mandated. You can create and enable a set of policies, group them in one or more policy categories, and then force all databases to subscribe to these policies.

If you don't assign a policy to a policy category, it is placed in the *Default* policy category, which always mandates a subscription from databases. Unlike other policy categories, the *Default* policy category cannot be changed to optionally allow subscriptions, so if you put a policy in this policy category and the policy is enabled and enforces compliance (*Changes are attempted, prevent out-of-compliance*), then all databases will have to comply. If you want the ability to optionally allow subscriptions to the policy category, you must add the policy to a policy category other than *Default* and then use the *Policy Category Management* to set the policy category mandate subscription option as false (unchecked).

## Checking and Enforcing Compliance

As mentioned earlier in this chapter, not all policies can be set to enforce compliance, but you can check compliance, on change or on schedule. But what happens when you do a manual check and you find something that is out-of-compliance? What is the process to get the server, database, and so on, back in compliance?

Let's revisit an example from earlier in the chapter. You create a policy that states "SQL Mail and Database Mail should both be disabled". You set its execution mode to *On Demand* and leave the policy disabled. Now, how do you go about checking the policy?

First we are going to set the server configuration so that it is out-of-compliance by running the following Transact-SQL code.

```
--Run this first to see advanced options
EXEC sp_configure 'show advanced options', 1
RECONFIGURE WITH OVERRIDE

--Run this second to change the mail configuration
EXEC sp_configure 'Database Mail XPs', 1
EXEC sp_configure 'SQL Mail XPs', 0
RECONFIGURE WITH OVERRIDE
```

Listing 4

Now your server will fail the compliance check. Next, we check the compliance of the policy by right-clicking on the SQL Server Instance in *Object Explorer* and choosing Policies, then View, as shown in Figure 1-8.

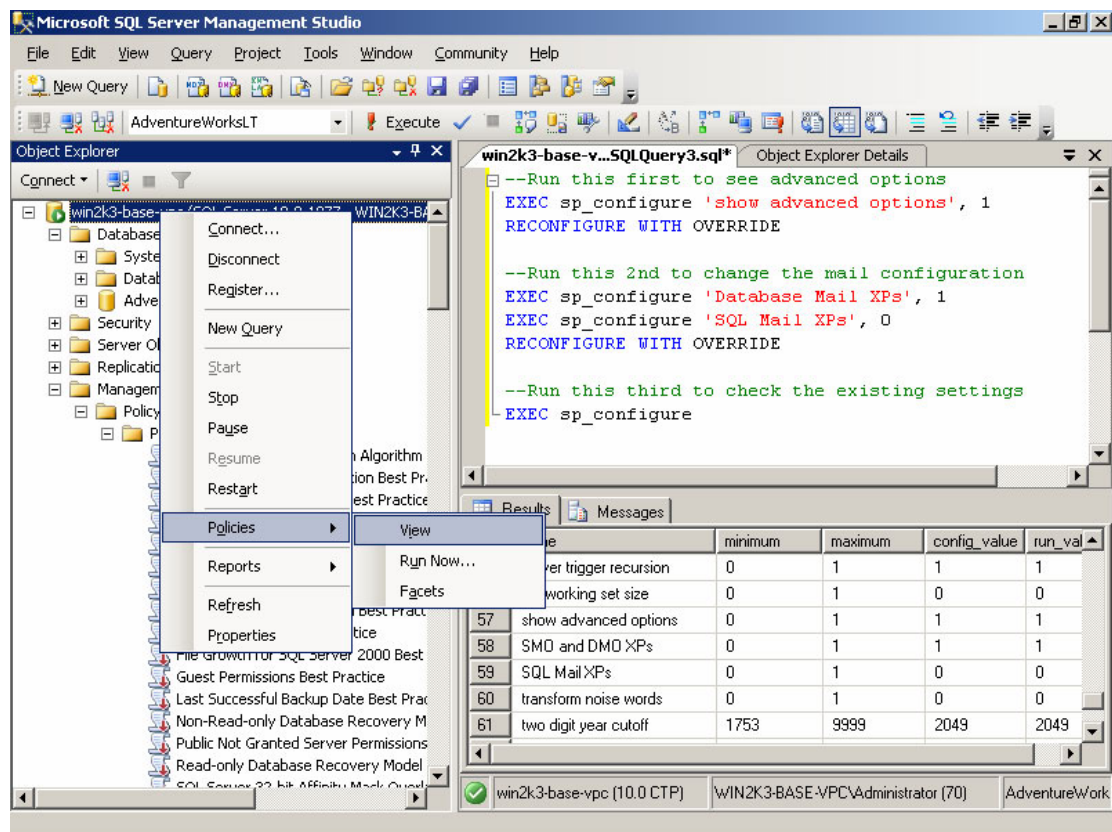


Figure 1-8

### Server View Policies Context Menu.

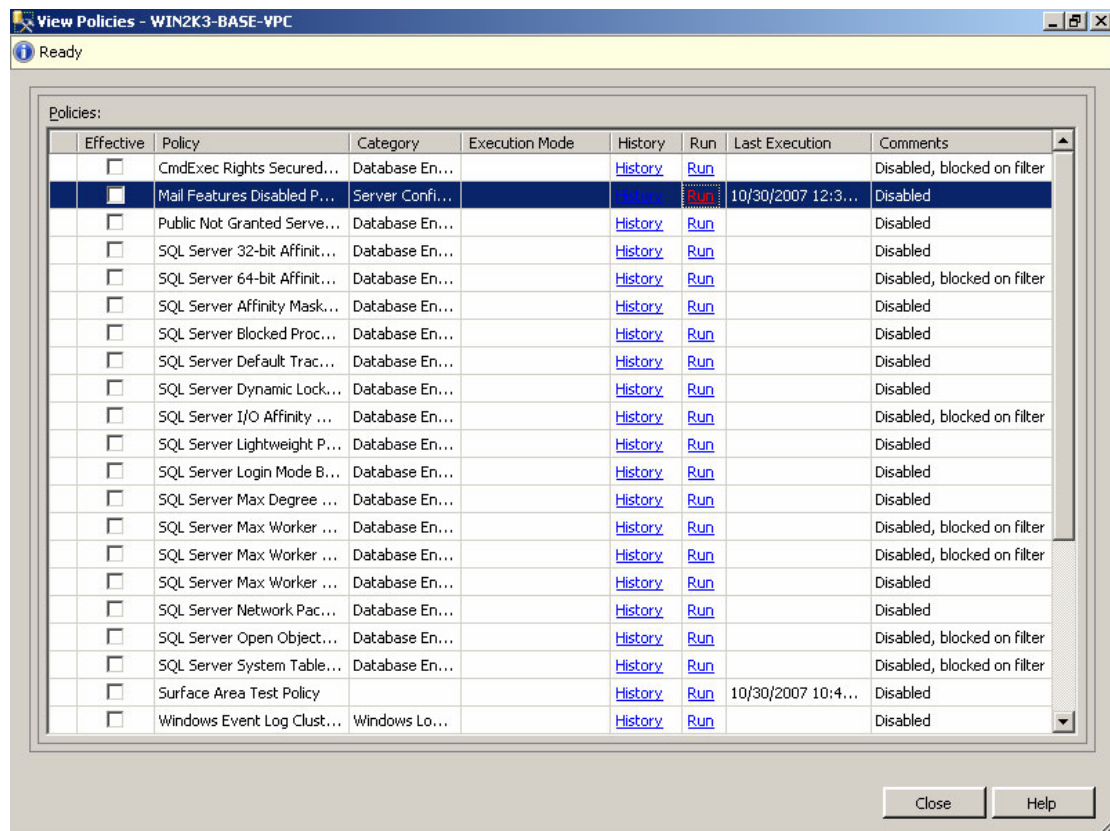


Figure 1-9

### View Policies Dialog Box.

This will bring up the View Policies dialog box shown in Figure 1-9, where you can view information about the policies, including whether or not the policy is enabled (Effective), its category, last execution, and comments. But here you can also click to see a history of the policy and to run the policy and check compliance.

**Note** If you want to see a history of all policies, you can use the context menu of the Policies node in Object Explorer and choose View history.

As shown in Figure 1-10, clicking Run reveals that the server is out-of-compliance with this policy (as expected since we purposefully broke compliance a few moments ago).

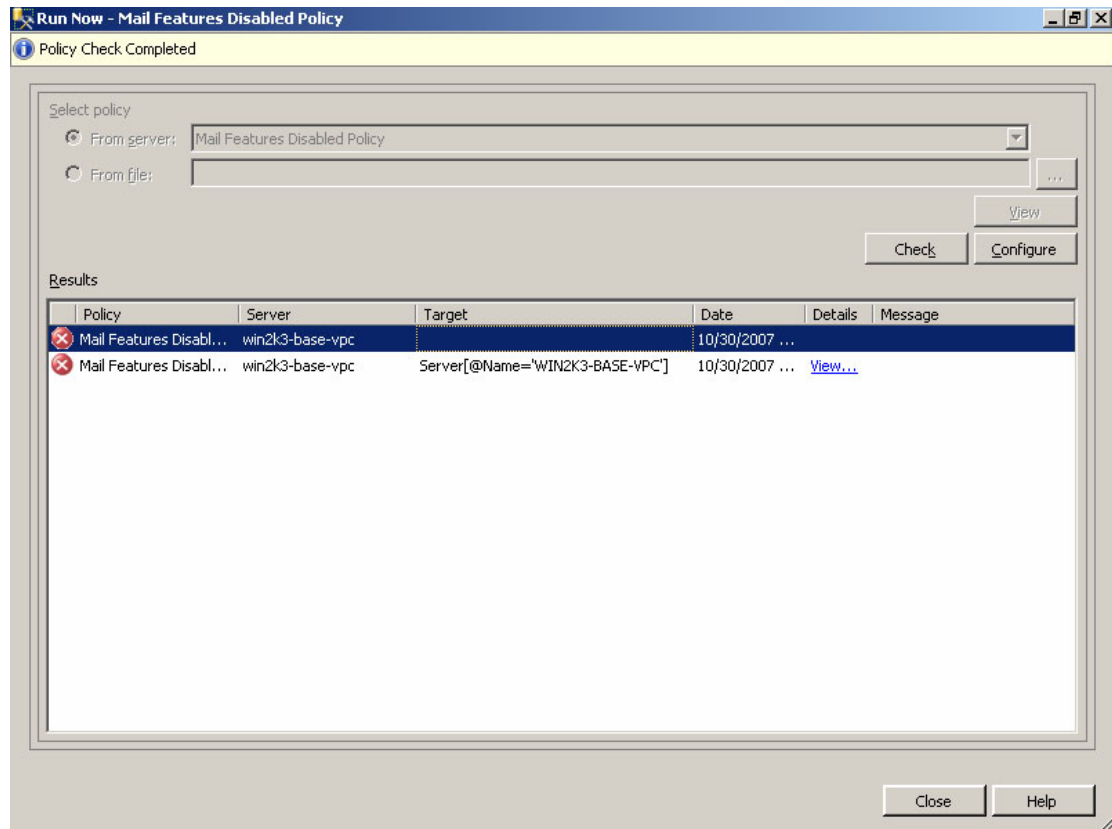


Figure 1-10

**Run Now Dialog Box for the Mail Features Disabled Policy.**

Clicking the Check button will cause the policy to be checked again and will result in the same thing—a policy that is out-of-compliance. But clicking the “Configure” button will simply fix the problem, as shown in Figure 1-11.

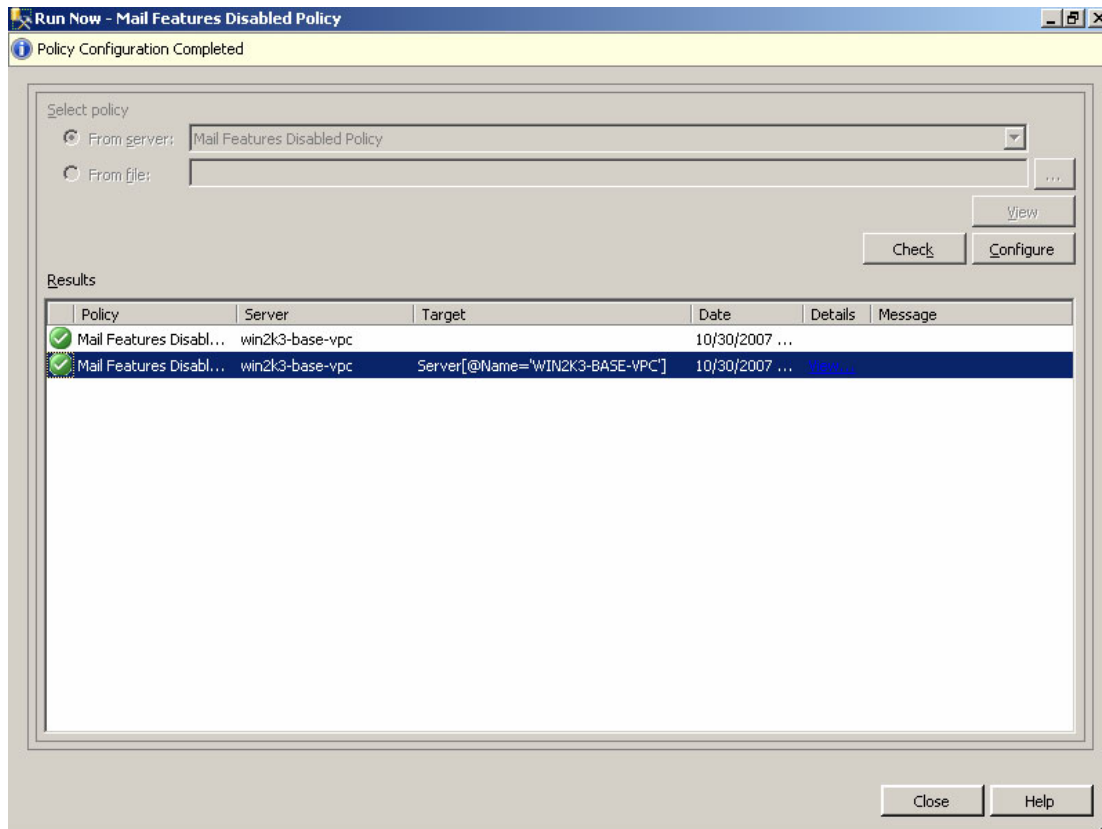


Figure 1-11

**Mail Features Disabled Policy now in Compliance.**

Indeed, a single click and you can get the server back in compliance. It makes the appropriate changes to the configuration and reruns the policy check, resulting in a policy that is now compliant.

**Note** You can also view and test (check) individual policies by right-clicking the policy located under the Policy node in Object Explorer, or you can view policies for other objects in Object Explorer, such as a database or a table.

**The DMF in Practice**

At this point, we will run through a simple example of using the DMF to enforce naming conventions on our tables, stored procedures, and functions. Here is a list of conditions you will need to create:

Condition Name	Facet	Expression
Stored Procedure Name	Stored Procedure	@Name NOT LIKE 'sp[_]%'
Table Name	Table	@Name NOT LIKE 'tbl%'
Function Name	User Defined Function	@Name LIKE 'fn%'

The next step is to create three corresponding policies that are all part of the same category named "Naming Conventions". All policies should use the default setting for targets, servers, enabled, and set to check compliance when *Changes are attempted, prevent out-of-compliance*.

Policy Name	Condition
Stored Procedure Name Policy	Stored Procedure Name
Table Name Policy	Table Name
Function Name Policy	Function Name

Figure 1-12 shows an example of the Stored Procedure name Policy and its appropriate settings.

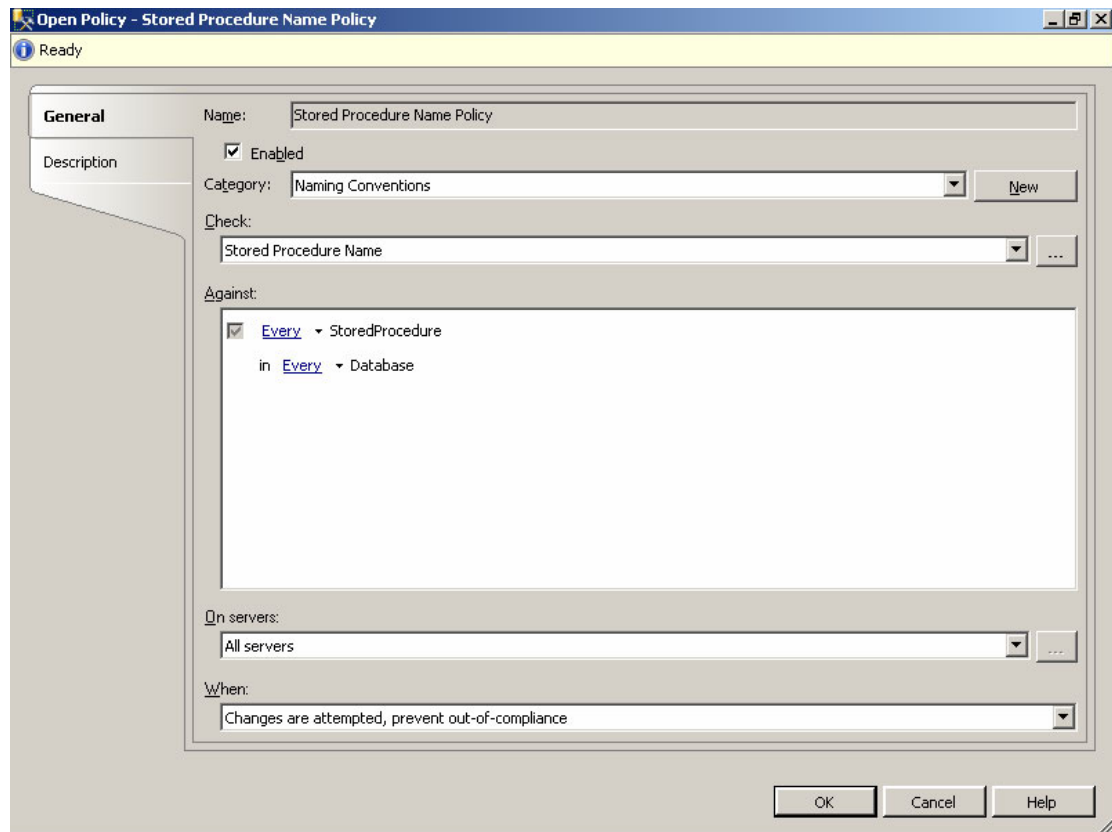


Figure 1-12

### Create New Condition Dialog Box.

By default, all new categories are set to mandate subscriptions, so using the AdventureWorksLT example database we can try to create the various objects, as shown in Listing 5.

```
CREATE PROCEDURE sp_test
AS
    SELECT 1 As [one]
GO

CREATE TABLE tblTest
(
    tbltestID int NOT NULL,
    Description varchar(100) NULL
)
GO

CREATE FUNCTION fTest (@i INT)
RETURNS INT
AS
BEGIN
    RETURN @i * 2
END
GO
```

Listing 5

Running this script will result in the following:

```
win2k3-base-vpc(WIN2K3-BASE-VPC\Administrator):
Policy 'Stored Procedure Name Policy' has been violated by
'Server/Database[@Name='AdventureWorksLT']/StoredProcedure[@Name='sp_test' and
@Schema='dbo']'.
This transaction will be rolled back.
Policy description: ''
Additional help: '' : ''.
win2k3-base-vpc(WIN2K3-BASE-VPC\Administrator): Msg 3609, Level 16, State 1,
Procedure sp_syspolicy_dispatch_event, Line 50
The transaction ended in the trigger. The batch has been aborted.

win2k3-base-vpc(WIN2K3-BASE-VPC\Administrator):
Policy 'Table Name Best Practice' has been violated by
'Server/Database[@Name='AdventureWorksLT']/Table[@Name='tblTest' and
@Schema='dbo']'.
This transaction will be rolled back.
Policy description: ''
Additional help: '' : ''.
win2k3-base-vpc(WIN2K3-BASE-VPC\Administrator): Msg 3609, Level 16, State 1,
Procedure sp_syspolicy_dispatch_event, Line 50
The transaction ended in the trigger. The batch has been aborted.

win2k3-base-vpc(WIN2K3-BASE-VPC\Administrator):
Policy 'Function Name Policy' has been violated by
'Server/Database[@Name='AdventureWorksLT']/UserDefinedFunction[@Name='fTest' and
@Schema='dbo']'.
This transaction will be rolled back.
Policy description: ''
Additional help: '' : ''.
win2k3-base-vpc(WIN2K3-BASE-VPC\Administrator): Msg 3609, Level 16, State 1,
Procedure sp_syspolicy_dispatch_event, Line 50
The transaction ended in the trigger. The batch has been aborted.
```

Listing 6

You will notice that there is additional information such as “Policy description” which is simply an empty string. You can add this additional description and help information in the policy itself. Figure 1-13 shows an example of setting a description, help text, and URL for the “Stored Procedure Name Policy”.

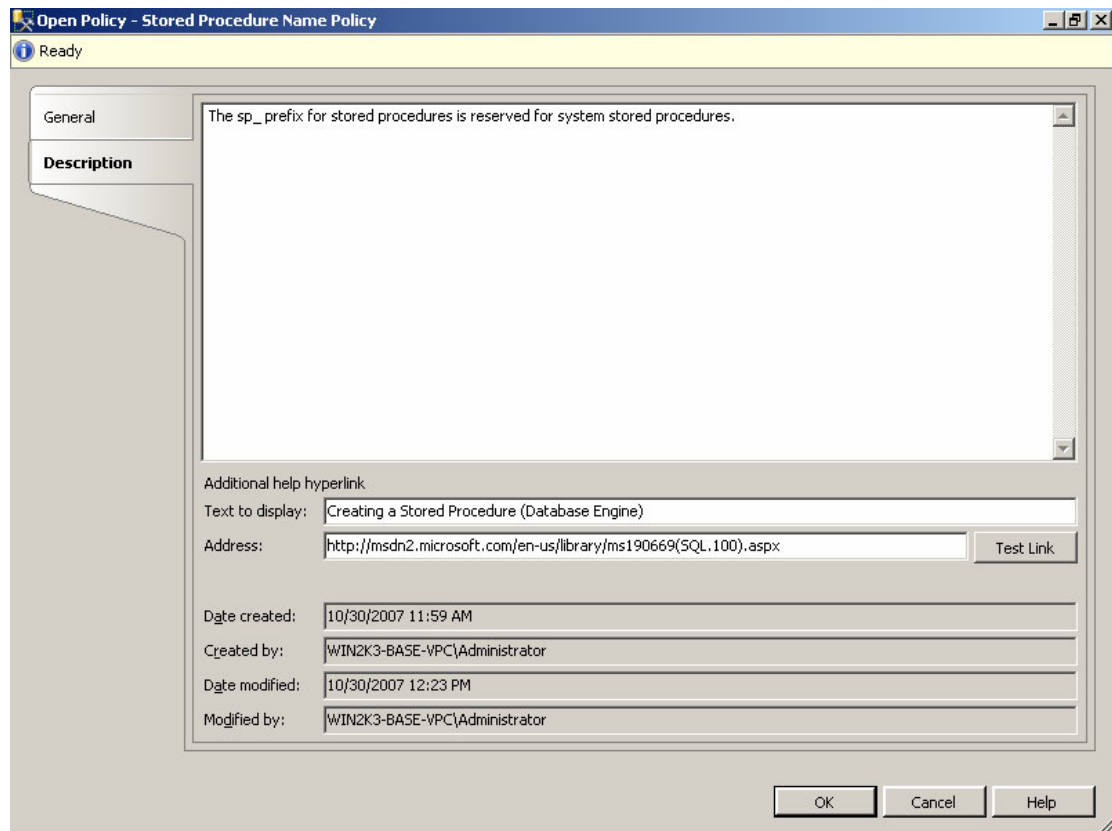


Figure 1-13

### Policy Description Settings

Using the information shown in Figure 1-13, change the “Stored Procedure Name Policy” description information and try to create the stored procedure again. The results will now show:

```
win2k3-base-vpc(WIN2K3-BASE-VPC\Administrator):
Policy 'Stored Procedure Name Policy' has been violated by
'Server/Database[@Name='AdventureWorksLT']/StoredProcedure[@Name='sp_test' and
@Schema='dbo']'.
This transaction will be rolled back.
Policy description: 'The sp_ prefix for stored procedures is reserved for system
stored procedures.'
Additional help: 'Creating a Stored Procedure (Database Engine)' :
'http://msdn2.microsoft.com/en-us/library/ms190669(SQL.100).aspx'.
win2k3-base-vpc(WIN2K3-BASE-VPC\Administrator): Msg 3609, Level 16, State 1,
Procedure sp_syspolicy_dispatch_event, Line 50
The transaction ended in the trigger. The batch has been aborted.
```

Listing 7

## Summary

The Declarative Management Framework makes light of the work previously required to keep servers, databases, and an assortment of objects in compliance with company policies. There are other features which have not yet been discussed, such as the Advanced Edit capability for Conditions, which allows you to create custom expressions using the available facet properties and a variety of built-in functions (including the ability to dynamically execute T-SQL). I hope that this gives you enough to become familiar with the workings of the DMF and to continue increasing your knowledge of the DMF.

SQL Server 2008 Books Online has additional examples and step-by-step instructions on how to create several different policies. I recommend reading through these and trying them out as they can give types of examples other than those covered in this chapter.

## Chapter 11

# Transact-SQL Enhancements

## Variable Declaration and Assignment

A long time ago, I learned that Transact-SQL was not like other programming languages. And for a long time, I accepted the fact that when I declared a variable, all I could do was to declare it. Sure, stored procedure and user-defined function parameters could be declared and set to a default, but that simply didn't happen for inline variable declarations in T-SQL. That is, until now...

Indeed, it is true. You can now declare and assign a variable in one statement, so instead of writing this code...

```
DECLARE
    @someIntVal INT,
    @someStringVal varchar(100)
SET @someIntVal = 1
SET @someStringVal = 'One'
```

you can now write this code...

```
DECLARE
    @someIntVal INT = 1,
    @someStringVal varchar(100) = 'One'
```

I know it seems like such a simple change, but believe me, it adds up over time. For example, look at this set of variable declarations and assignments from the `aspnet_Membership_CreateUser` stored procedure:

```
DECLARE @ApplicationId uniqueidentifier
SELECT @ApplicationId = NULL

DECLARE @NewUserId uniqueidentifier
SELECT @NewUserId = NULL

DECLARE @IsLockedOut bit
SET @IsLockedOut = 0

DECLARE @LastLockoutDate datetime
SET @LastLockoutDate = CONVERT( datetime, '17540101', 112 )

DECLARE @FailedPasswordAttemptCount int
SET @FailedPasswordAttemptCount = 0

DECLARE @FailedPasswordAttemptWindowStart datetime
SET @FailedPasswordAttemptWindowStart = CONVERT( datetime, '17540101', 112 )

DECLARE @FailedPasswordAnswerAttemptCount int
SET @FailedPasswordAnswerAttemptCount = 0

DECLARE @FailedPasswordAnswerAttemptWindowStart datetime
SET @FailedPasswordAnswerAttemptWindowStart = CONVERT( datetime, '17540101', 112 )
```

```
DECLARE @NewUserCreated bit

DECLARE @ReturnValue int
SET @ReturnValue = 0

DECLARE @ErrorCode int
SET @ErrorCode = 0

DECLARE @TranStarted bit
SET @TranStarted = 0
```

This above approach allows you to see each variable declaration and assignment together (a stylistic choice). Particularly for such coding techniques, the assign the new declare and assign ability in Microsoft SQL Server 2008 has great potential, as shown here.

```
DECLARE
    @ApplicationId uniqueidentifier = NULL,
    @NewUserId uniqueidentifier = NULL,
    @IsLockedOut bit = 0,
    @LastLockoutDate datetime = CONVERT(datetime, '17540101', 112),
    @FailedPasswordAttemptCount int = 0,
    @FailedPasswordAttemptWindowStart datetime = CONVERT(datetime, '17540101',
112),
    @FailedPasswordAnswerAttemptCount int = 0,
    @FailedPasswordAnswerAttemptWindowStart datetime
        = CONVERT(datetime, '17540101', 112),
    @NewUserCreated bit,
    @ReturnValue int = 0,
    @ErrorCode int = 0,
    @TranStarted bit = 0
```

This “declare and assign” ability can be used with variables, such that the following two declarations are also valid:

```
DECLARE @x int = 10
DECLARE @y int = @x + 1
```

But mind you, since the declaration of @y references @x, it cannot be in the same declaration, as shown here:

```
-- Not allowed
DECLARE @x int = 10, @y int = @x + 1
```

I expect that you are now speechless, and so I will show you the second new feature for variable assignment in SQL Server 2008. I give you compound assignment:

```
DECLARE @n int = 10
SET @n += 1
SELECT @n AS [n]
```

As you expect, this will return a value of 11. Compound assignment allows you to skip the repetition of a variable that is being assigned based on itself such that `SET @x = @x + 1` is the same as `SET @x += 1`. There is a total of eight compound assignment operators.

Operator	Description	Operator	Description
+=	Add and assign	-=	Subtract and assign
*=	Multiply and assign	/=	Divide and assign
%=	Modulo and assign	&=	Bitwise AND and assign
^=	Bitwise XOR and assign	=	Bitwise OR and assign

Compound assignment is like any other form of assignment and cannot be done inline in a statement. So, statements such as the following are not allowed.

```
DECLARE @a int = 3
--This line of code will fail
SET @a *= @a += 6
```

You can, however, assign the same variable multiple times using a `SELECT` statement, as shown here.

```
DECLARE @a int = 5;
SELECT @a *= @a, @a += @a;
SELECT @a;
```

And so, you must be asking yourself “what is the final value of @a?” As you would expect, the result is 50. First the compound multiplication is executed, changing @a value to 25 (5 \* 5) and then it is added to itself, resulting in 50 (25 + 25). As a matter of fact, when you have multiple assignments in a `SELECT` statement, they seem to always be done from left to right, regardless of the operation being performed, which makes perfect sense being that there is no order of operations involved since these are individual assignments, so although you might be inclined to think that the following code would also return a value of 50, it actually returns 100.

```
DECLARE @a int = 5;
SELECT @a += @a, @a *= @a;
SELECT @a;
```

Just in case this isn't clear why, keep in mind that that code is the same as this code.

```
DECLARE @a int = 5;
SELECT @a = @a + @a, @a = @a * @a;
SELECT @a;
```

Assignment occurs from left to right, regardless of what actual expression is being used to determine the resulting value. What I am trying to convey is that you should not confuse compound assignment with other T-SQL operators—these are simply still assigning values and do not have an order of operation.

## Row Constructors

Here is an excerpt of a script that installs the “msdb” system database.

```
INSERT INTO dbo.syscategories (category_id, category_class, category_type, name)
VALUES ( 0, 1, 1, N'[Uncategorized (Local)]')
INSERT INTO dbo.syscategories (category_id, category_class, category_type, name)
VALUES ( 1, 1, 1, N'Jobs from MSX')
INSERT INTO dbo.syscategories (category_id, category_class, category_type, name)
VALUES ( 2, 1, 2, N'[Uncategorized (Multi-Server)]')
INSERT INTO dbo.syscategories (category_id, category_class, category_type, name)
VALUES ( 3, 1, 1, N'Database Maintenance')
INSERT INTO dbo.syscategories (category_id, category_class, category_type, name)
VALUES ( 4, 1, 1, N'Web Assistant')
INSERT INTO dbo.syscategories (category_id, category_class, category_type, name)
VALUES ( 5, 1, 1, N'Full-Text')
INSERT INTO dbo.syscategories (category_id, category_class, category_type, name)
VALUES ( 6, 1, 1, N'Log Shipping')
INSERT INTO dbo.syscategories (category_id, category_class, category_type, name)
VALUES ( 7, 1, 1, N'Database Engine Tuning Advisor')
INSERT INTO dbo.syscategories (category_id, category_class, category_type, name)
VALUES (98, 2, 3, N'[Uncategorized]')
INSERT INTO dbo.syscategories (category_id, category_class, category_type, name)
VALUES (99, 3, 3, N'[Uncategorized]')
```

Some people prefer to use a SELECT with UNION ALL to do the same operation, as shown here:

```
INSERT INTO dbo.syscategories (category_id, category_class, category_type, name)
SELECT 0, 1, 1, N'[Uncategorized (Local)]' UNION ALL
SELECT 1, 1, 1, N'Jobs from MSX' UNION ALL
SELECT 2, 1, 2, N'[Uncategorized (Multi-Server)]' UNION ALL
SELECT 3, 1, 1, N'Database Maintenance' UNION ALL
SELECT 4, 1, 1, N'Web Assistant' UNION ALL
SELECT 5, 1, 1, N'Full-Text' UNION ALL
SELECT 6, 1, 1, N'Log Shipping' UNION ALL
SELECT 7, 1, 1, N'Database Engine Tuning Advisor' UNION ALL
SELECT 98, 2, 3, N'[Uncategorized]' UNION ALL
SELECT 99, 3, 3, N'[Uncategorized]'
```

And now, in SQL Server 2008, you can simply execute the following statement to achieve the same results.

```
INSERT INTO dbo.syscategories (category_id, category_class, category_type, name)
VALUES
( 0, 1, 1, N'[Uncategorized (Local)]'),
( 1, 1, 1, N'Jobs from MSX'),
( 2, 1, 2, N'[Uncategorized (Multi-Server)]'),
( 3, 1, 1, N'Database Maintenance'),
( 4, 1, 1, N'Web Assistant'),
( 5, 1, 1, N'Full-Text'),
( 6, 1, 1, N'Log Shipping'),
( 7, 1, 1, N'Database Engine Tuning Advisor'),
(98, 2, 3, N'[Uncategorized]'),
(99, 3, 3, N'[Uncategorized]')
```

The difference is simply amazing. I was a big fan of using SELECT statements with UNION ALL to generate a row set, and now it is very apparent to me that this row constructor feature will further simplify the scripting of data, the creation of demos, and so on. Of course, there is row constructor ability besides inserting multiple rows of data. Back in an earlier chapter, I was querying the Declarative Management Framework system tables and views, as shown here:

```

;WITH AutomatedPolicyExecutionMode (ModeId, ModeName)
AS
(SELECT *
 FROM (VALUES
      (0, 'On demand'),
      (1, 'Enforce Compliance'),
      (2, 'Check on change and log'),
      (4, 'Check on schedule and log')
 ) AS EM(ModeId, ModeName)
)
SELECT
    pmf.[management_facet_id] AS FacetID
    , pmf.[name] AS FacetName
    , APEM.[ModeName]
FROM syspolicy_management_facets AS pmf
    INNER JOIN AutomatedPolicyExecutionMode AS APEM
        ON pmf.[execution_mode] & APEM.[ModeId] = APEM.[ModeId]
ORDER BY pmf.[name], APEM.[ModeId]

```

Since there was no metadata for the various execution modes, I created it using the row constructor ability combined with a Common Table Expression (CTE) to allow me to use the newly constructed row data in a join with another table. Again, this could have been done using a SELECT statement with UNION ALL; and it behaves in exactly the same way. The only difference is that as the number of generated rows grows, this syntax will require less typing, and less space in script.

## Merge

Back before SQL Server 2005 was released, there was talk of a feature that was known by the name “Merge” or “Upsert,” which was the ability to insert and update a table’s data in a single DML statement. Alas, the feature didn’t make it into the RTM of SQL Server 2005; much to my satisfaction and pleasure, however, it has made it into SQL Server 2008.

The new MERGE statement has the ability to insert new data into a target table, and update or delete existing data in that target table, directly or by means of a view, using a single T-SQL statement.

The abridged syntax of MERGE is as follows:

```

MERGE <target_table> [ AS table_alias ]
USING <table_source>
ON <search_condition>
[WHEN MATCHED [ AND <search_condition>]
    THEN {UPDATE... | DELETE} ]
[WHEN TARGET NOT MATCHED [ AND <search_condition>]
    THEN INSERT... ]
[WHEN SOURCE NOT MATCHED [ AND <search_condition>]
    THEN {UPDATE... | DELETE} ]
;

```

**Note** Although using a semicolon as a statement terminator is not officially required by most T-SQL statements, the MERGE statement does require its use.

The MERGE statement initially specifies a target table, which can actually be a table or a view (see “Modifying Data Through a View” in SQL Server 2008 Books Online for details about modifying through a view). The USING clause specifies the source of the data against which you will be comparing the target data. The ON clause specifies the basis for the comparison between the target and the source.

For example, you may be using the Customer table as your target and the NewCustomer table as the source, where NewCustomer contains a batch of customers that need to be inserted or potentially updated in your existing Customer table data. If the customer needs to be updated, the primary key value of the customer, CustomerID, is used to match the updated customer data to the existing customer, as shown here:

```
MERGE Customer AS c
USING NewCustomer AS nc
ON c.CustomerID = nc.CustomerID
WHEN TARGET NOT MATCHED THEN
    INSERT (FirstName, LastName, CompanyName, EmailAddress, Phone)
    VALUES (nc.FirstName, nc.LastName, nc.CompanyName, nc.EmailAddress,
nc.Phone)
WHEN MATCHED THEN
    UPDATE SET
        FirstName = nc.FirstName,
        LastName = nc.LastName,
        CompanyName = nc.CompanyName,
        EmailAddress = nc.EmailAddress,
        Phone = nc.Phone;
```

In this preceding example, new customers (which would have a CustomerID of 0 or NULL) would get inserted into the Customer table, and existing customers (matched on CustomerID) would always be updated, regardless if the data in the target and source was different or not.

## The WHEN Clauses

The key to MERGE are the three WHEN clauses used to determine “when” the various actions should take place.

### WHEN MATCHED

The WHEN MATCHED clause is used to find matches between the target and source tables and can either perform an UPDATE or DELETE against the target table. There are a few rules that you need to keep in mind when using this clause.

- When performing an UPDATE with this clause, the match condition must only return one row from the source table, because the MERGE statement cannot perform the same DML operation on a single row of data in the target more than one time. If the WHEN MATCHED condition returns more than one row and an UPDATE is used, an error will result.

- WHEN MATCHED can be used at most two times in the MERGE statement. If used twice the following rules also apply:
  - The two clauses are processed in order.
  - One clause must UPDATE and the other one must DELETE (order is not important).
  - The second WHEN MATCH clause is checked only if the first is not satisfied.
  - The first WHEN MATCHED clauses must specify additional criteria. If you attempt to execute without specifying additional search criteria for the first WHEN MATCHED clause, you will receive an error.

**Note** Although the first clause is required to have additional criteria, it doesn't prevent you from using the same criteria for both WHEN MATCHED clauses. If you do use the same criteria for both WHEN MATCHED clauses, the second one will never get processed since the second clause only gets checked when the first one does not match the specified criteria, and if they have the same criteria, the second would also not match.

## WHEN [TARGET] NOT MATCHED

WHEN TARGET NOT MATCHED (the keyword TARGET is optional, although I suggest being explicit and using it) is used to determine if there are rows in the source table that don't exist in the target table. In other words, it is used to find rows that might need to be inserted into the target table from the source table.

- This is the only WHEN clause that can INSERT data into the source table.
- There can be at most one WHEN TARGET NOT MATCHED clause in a MERGE statement.
- Additional criteria is optional.

## WHEN SOURCE NOT MATCHED

This clause is used to find rows in the target table that do not exist in the source table.

Again, some rules to abide by when using WHEN SOURCE NOT MATCHED:

- WHEN SOURCE NOT MATCHED can be used at most two times in the MERGE statement. If used twice the following rules also apply:
  - The two clauses are processed in order.
  - One clause must UPDATE and the other one must DELETE (order is not important).
  - The second clause is checked only if the first is not satisfied.
  - The first clause must specify additional criteria. If you attempt to execute without specifying additional search criteria for the first clause, you will receive an error.

**Note** Although the first clause is required to have additional criteria, it doesn't prevent you from using the same criteria for both WHEN SOURCE NOT MATCHED clauses. If you do use the same criteria for both WHEN SOURCE NOT MATCHED clauses, the second one will never get processed since the second clause only gets checked when the first one does not match the specified criteria, and if they have the same criteria, the second would also not match.

## Other Notes on All Matching Clauses

INSTEAD OF triggers defined on target table work in the same fashion as they always have, and do not actually modify the underlying table, but rather, rely on the INSTEAD OF trigger to do that work (by the now populated *inserted* and *deleted* trigger tables). Also, if you define an INSTEAD OF trigger for any action on the target table, there must be an INSTEAD OF trigger for all actions on the target table.

In order to achieve the correct matching, the MERGE statement does outer joins between the target and source data as needed. When using the WHEN [TARGET] NOT MATCHED clause, an outer join occurs from the source table to the target table. This means that for all rows from the source table, MERGE checks to see if a match exists in the target, which means that all rows from the source are returned. The converse is true when using the WHEN SOURCE NOT MATCHED clause, and all rows from the target table are returned in order to perform an outer join from the target table to the source table.

Therefore, if you use both the WHEN [TARGET] NOT MATCHED and the WHEN SOURCE NOT MATCHED clauses in a single MERGE statement, an outer join is used in both directions resulting in the use of a full outer join.

## MERGE Exemplified

Let's go through a simple example using the new AdventureWorksLT database sample. This new sample database contains customers and their associated sales data. For reporting purposes, I want to create a flat view of customers and sales total and last sales date, and so I first create a table to hold that information, as shown here:

```
USE AdventureWorksLT;
GO
CREATE TABLE SalesLT.CustomerTotals
(
    CustomerID int PRIMARY KEY,
    LastOrderDate datetime,
    SalesTotal money
);
GO
```

From here, I will do a very simple merge between the Customer table and the new CustomerTotals table.

```
MERGE SalesLT.CustomerTotals AS ct
USING SalesLT.Customer AS c
ON c.CustomerID = ct.CustomerID
WHEN TARGET NOT MATCHED THEN
    INSERT (CustomerID)
    VALUES (c.CustomerID);
```

This MERGE statement looks to see what rows in Customer (the source) do not have a corresponding match in CustomerTotals (the target). Since the CustomerTotal table is empty, all rows in Customer should meet the criteria and you should see a message that 440 rows were affected (assuming you haven't deleted any customers yet). Running it a

second time will result in zero (0) rows being affected, since all rows will now match between target and source (since you just inserted them from the source to the target).

Now, we will use data from the SalesOrderHeader table to update the CustomerTotals table with sales total and last order date data, as shown here:

```
WITH CustSales AS
(
    SELECT
        CustomerID,
        MAX(OrderDate) as MaxOrderDate,
        SUM(TotalDue) as TotalDueTotal
    FROM SalesLT.SalesOrderHeader
    GROUP BY CustomerID
)
MERGE SalesLT.CustomerTotals AS ct
USING CustSales
ON CustSales.CustomerID = ct.CustomerID
WHEN MATCHED THEN
    UPDATE SET
        LastOrderDate = cs.MaxOrderDate,
        SalesTotal = cs.TotalDueTotal
WHEN SOURCE NOT MATCHED THEN
    DELETE
WHEN TARGET NOT MATCHED THEN
    INSERT (CustomerID, LastOrderDate, SalesTotal)
    VALUES (CustSales.CustomerID, CustSales.MaxOrderDate,
    CustSales.TotalDueTotal);
```

So what is this MERGE statement actually doing? The MERGE is updating the last sales date and total sales amount for any customer that exists in both the target (CustomerTotals) and the source (a CTE of aggregated customer sales data). The MERGE is also deleting any customers from the target table that don't have any associated sales (doesn't exist in the customer sales CTE); and it is adding new customers into the target table if they don't already exist in that table but do exist in the source.

As you may now realize, the first step to populate the CustomerTotals table was unnecessary since this statement would also populate the CustomerID values (without the overhead of populating them all and then removing a majority of them). What you may not have noticed is that although this MERGE seems to be exactly what you want, it is very inefficient. Stop for a moment and examine the code and see if you can spot the inefficiency. I'll wait for you...

## Optimizing MERGE

Now, you have pondered for at least a few moments about the potential performance problem the last MERGE statement could cause. So where is the inefficiency? It is in the WHEN MATCHED clause. Imagine if you had millions of active customers and you ran that MERGE statement that checked the sales order header data against the customer totals data. Every time you execute that MERGE statement, you would be updating millions of records with the same data—not very efficient. A better choice would be to only update those records that have changed. And so, a small change to the WHEN MATCHED clause can make a world of difference, as shown here:

```

WITH CustSales AS
(
    SELECT
        CustomerID,
        MAX(OrderDate) as MaxOrderDate,
        SUM(TotalDue) as TotalDueTotal
    FROM SalesLT.SalesOrderHeader
    GROUP BY CustomerID
)
MERGE SalesLT.CustomerTotals AS ct
USING CustSales
ON CustSales.CustomerID = ct.CustomerID
WHEN MATCHED AND LastOrderDate != CustSales.MaxOrderDate THEN
    UPDATE SET
        LastOrderDate = cs.MaxOrderDate,
        SalesTotal = cs.TotalDueTotal
WHEN SOURCE NOT MATCHED THEN
    DELETE
WHEN TARGET NOT MATCHED THEN
    INSERT (CustomerID, LastOrderDate, SalesTotal)
    VALUES (CustSales.CustomerID, CustSales.MaxOrderDate,
    CustSales.TotalDueTotal);

```

Now, only if the Customer IDs match and the date of the last order in the sales order data is different from the date of the last order in the customer totals data will the CustomerTotals table be updated. The point of this exercise is to make you aware that you should test your code and ensure that you are only doing the operations that you need to do in order to insert, update, or delete your target data.

Still another alternate approach would be only to update since the last known change (which also means we wouldn't delete missing records since our source would only have a subset of customers), as shown here:

```

WITH cs AS
(SELECT CustomerID, MAX(OrderDate) as MaxOrderDate, SUM(TotalDue) as
TotalDueTotal
FROM SalesLT.SalesOrderHeader
WHERE OrderDate >
ISNULL((SELECT MAX(LastOrderDate) FROM SalesLT.CustomerTotals), 0)
GROUP BY CustomerID
)
MERGE SalesLT.CustomerTotals AS ct
USING cs
ON cs.CustomerID = ct.CustomerID
WHEN MATCHED THEN
    UPDATE SET
        LastOrderDate = cs.MaxOrderDate,
        SalesTotal += cs.TotalDueTotal
WHEN TARGET NOT MATCHED THEN
    INSERT (CustomerID, LastOrderDate, SalesTotal)
    VALUES (cs.CustomerID, cs.MaxOrderDate, cs.TotalDueTotal);

```

In the end, how you use and optimize the use of MERGE will be entirely up to what the business requires, or in consulting words—it depends.

## User-Defined Table Type and Table-Valued Parameters

There have been numerous ways that people have used to pass the equivalent of a table of data into a stored procedure, user-defined function, and so on. Delimited strings and XML are two common implementations, although one could also use a custom CLR type, fixed-length binary data, and so on. Long story short—you could pass almost everything except a table. At this point, I'm sure you will not be surprised to learn that SQL Server 2008 now has the ability to pass tables as parameters. But in order to be able to pass table-valued parameters, we first need to go back and learn about new user-defined table types in our database.

### User-Defined Table Type

You can now declare a type that represents a table structure. For example, suppose you frequently need to use the following structure in your T-SQL code:

```
DECLARE @NewCustomer TABLE
(
    [CustomerID] int NULL,
    [FirstName] varchar(50) NOT NULL,
    [LastName] varchar(50) NOT NULL,
    [CompanyName] [nvarchar](128) NULL
)
```

After repeating this code many times in a variety of stored procedures and user-defined functions, you begin to wish for an easier way to make these declarations. And of course, there is a better way—use a user-defined table type. Here is the definition of the `NewCustomer` table:

```
CREATE TYPE [NewCustomer] AS TABLE
(
    [CustomerID] int NULL,
    [FirstName] varchar(50) NOT NULL,
    [LastName] varchar(50) NOT NULL,
    [CompanyName] [nvarchar](128) NULL
)
```

Once you have defined the table type, you can simply use it when declaring variables and parameters, as shown here:

```
DECLARE @NewCustomer AS [NewCustomer]
```

This declaration is equivalent to the `DECLARE @NewCustomer TABLE` code found in the beginning of this section without all the extra typing required for each subsequent declaration.

Now of course, what would a new feature be without a few caveats. For example, you cannot define a column as a user-defined table type—that would be the equivalent of nesting tables, which cannot be done. There are other things that are required when using user-defined table types, such as:

- Default values are not allowed in the table definition.
- Primary Key columns must be on a persisted column that does not allow null values.
- Check constraints cannot be done on nonpersisted computed columns.
- Nonclustered indexes are not allowed. If, however, you create a [nonclustered] primary key or unique constraint (both of which are allowed), you will be creating a nonclustered index behind the scenes since SQL Server uses an index to enforce unique and primary key constraints.

**Note** You cannot ALTER a user-defined table type. Once created, it must be dropped and re-created in order to make any changes. This can become cumbersome if the user-defined table type has been used as a parameter in a stored procedure or user-defined function, since it can now also not be dropped. This means that you will need to do some planning instead of haphazardly using user-defined table types.

## Table-Valued Parameters

Once you have defined the necessary user-defined table types, you can then use them in your T-SQL code, as variables (shown above) or as parameters, as shown here:

```
CREATE PROCEDURE prCustomerInsertOrUpdate
    @NewCustomer [NewCustomer] READONLY
AS
MERGE Customer AS c
USING @NewCustomer AS nc
ON c.CustomerID = nc.CustomerID
WHEN TARGET NOT MATCHED THEN
    INSERT (FirstName, LastName, CompanyName, EmailAddress, Phone)
    VALUES (nc.FirstName, nc.LastName, nc.CompanyName, nc.EmailAddress,
nc.Phone)
WHEN MATCHED THEN
    UPDATE SET
        FirstName = nc.FirstName,
        LastName = nc.LastName,
        CompanyName = nc.CompanyName,
        EmailAddress = nc.EmailAddress,
        Phone = nc.Phone;
```

Notice how create a stored procedure that accepts a customer record as a parameter and then uses the MERGE statement to either insert or update the record accordingly. You may have noticed that the parameter uses the READONLY keyword. This is a requirement for table-valued parameters, which means you cannot modify a table-valued parameter using DML (which also means that it cannot be an OUTPUT parameter).

This static state of the parameter value does offer some benefits, though, such as no need to acquire locks when initially populating from the client. They also do not cause statements to recompile. And unlike other parameter types that may contain a set of data,

table-valued parameters are strongly typed. Yes, XML can also be strongly typed, but strongly-typed XML is generally less efficient to process when compared to the table-valued parameter.

However, table-valued parameters do have some minor restrictions. For example, statistics are not maintained for table-valued parameter columns. Table-valued parameters also have the same set of restrictions that table variables have, such that you cannot SELECT INTO them, and you cannot use them with an INSERT EXEC statement.

When sending one or a few rows to be used by a procedure or user-defined function, table-valued parameters are a great choice and with few exceptions, the best for performance. But when are the best times to use a table-valued parameter, versus other features such as BULK INSERT?

When calling from a remote client (middle tier or rich client), more often than not, using a table-valued parameter is the best choice, with one exception: If you need to insert more than a thousand rows (this number is not exact and should be used as a guideline and not a strict metric value) and you are directly inserting the data and not performing any complex business logic, then you should consider using BULK INSERT instead of table-valued parameters. On the other hand, if you are loading data into SQL Server from a file on the server, then you are only best served by a table-valued parameter when you are inserting less than a thousand rows (again, this number is not strict) and there is complex business logic required before inserting the data.

See SQL Server Books Online for more details about bulk loading data into SQL Server.

## Summary

There are some phenomenal new features available in T-SQL in SQL Server 2008. All of them are designed to make your programming experience better, and to give you more flexibility when writing code in T-SQL, and to give a potentially better performance. And if that wasn't enough, there are other new features that haven't been discussed here and for several of the features that were discussed, a completely new world to start exploring.