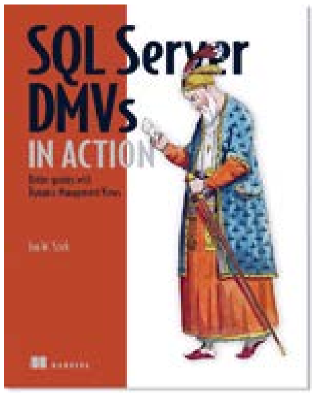

Looping over Databases on a Server

Often, servers contain many databases, and the server's resources are shared between these databases. This means that no matter how optimized a given database is, it could still perform suboptimally because another database on the server is running poorly and hogging the server's shared resources. Because of that, we should consider all databases on the server when we inspect the Dynamic Management Views (DMVs) with the goal of improving performance.

DMVs report data at a server or database level of detail. For example, at the server level, the DMV sys.dm_os_wait_stats reports the causes of all the waits on the server irrespective of individual databases. But, sometimes we need to obtain data at the database level of detail, for example, by joining DMV data with database level data contained in the catalog views. This typically provides us with richer data. As another example, we may want to retrieve index information from the catalog view sys.indexes; such information includes index name and type of index.

The data held by sys.indexes is located in each database on the server. In order to extract this index data to compare the data across all databases on the server, we need to loop around all the databases on the server. Luckily, Microsoft provides a stored procedure for this operation, the relatively little documented sp_MSForEachDB.

As an example of its use, we can print out the name of each of the databases on the server using the following command:

```
EXEC sp_MSForEachDB 'PRINT ''?'';'
```

Looping over the tables in a database

As an interesting side note, Microsoft also provides another related stored procedure, which loops over the tables within a given database. For example, to discover the names of each of the tables in a given database together with the number of rows they contain, you can run the below SQL query:

```
EXEC sp_MSForEachTable 'PRINT ''?''; SELECT ''?'' as [TableName], COUNT(*) AS
[RowCount] FROM ?;'
```

This pattern can be interesting for various maintenance tasks, such as marking the tables for recompilation (this effectively recompiles any stored procedures that use the tables) or updating a table's statistics.

Now back to our specific DMV use. We need to loop around the databases and join to the relevant database-specific sys.indexes catalog view so we can obtain some useful information about the index, including the index name and type. In the following example, we'll find the most used indexes to demonstrate the looping over the databases pattern.

Knowing about the most commonly used indexes is important, because they are the primary means of accessing data. We should ensure everything about these indexes is optimal, for example, that the statistics are up to date and have a good sampling percentage and the fill factor is such that we retrieve as much data as possible for every read. We should also ensure that the level of index fragmentation is low. Alternatively, a highly used index may indicate that a more optimized index is required but is not available, so SQL Server is forced to use whatever index is available, resulting in the apparent heavy index usage. Cross-checking this against the missing indexes, DMV will determine if this is true.

Running the code snippet given in listing 1 will loop over each database on the current server and report on the 10 most used indexes across all databases.

Listing 1 Looping over all databases on a server pattern

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT      #A
     DB_NAME() AS DatabaseName
     , SCHEMA_NAME(o.Schema_ID) AS SchemaName
     , OBJECT_NAME(s.[object_id]) AS TableName
     , i.name AS IndexName
     , (user_seeks + user_scans + user_lookups) AS [Usage]
     , user_updates
     , i.fill_factor
INTO #TempUsage
FROM sys.dm_db_index_usage_stats s
INNER JOIN sys.indexes i ON s.[object_id] = i.[object_id]
    AND s.index_id = i.index_id
INNER JOIN sys.objects o ON i.object_id = O.object_id
WHERE 1=2

EXEC sp_MSForEachDB 'USE [?];      #B
INSERT INTO #TempUsage
SELECT TOP 10
     DB_NAME() AS DatabaseName
     , SCHEMA_NAME(o.Schema_ID) AS SchemaName
     , OBJECT_NAME(s.[object_id]) AS TableName
     , i.name AS IndexName
     , (user_seeks + user_scans + user_lookups) AS [Usage]
     , user_updates
     , i.fill_factor
FROM   sys.dm_db_index_usage_stats s
INNER JOIN sys.indexes i ON s.[object_id] = i.[object_id]
                    AND s.index_id = i.index_id
INNER JOIN sys.objects o ON i.object_id = O.object_id
WHERE s.database_id = DB_ID()
    AND i.name IS NOT NULL
    AND OBJECTPROPERTY(s.[object_id], ''IsMsShipped'') = 0
ORDER BY [Usage] DESC'     #C

SELECT TOP 10 * FROM #TempUsage ORDER BY [Usage] DESC

DROP TABLE #TempUsage
```

#A Creating a temporary table to hold the results
#B Looping around all databases
#C Identifying most used indexes

The code snippet in listing 1 first creates a temporary table using a `WHERE 1=2` pattern. Often, we want to create a temporary table to hold the results of any transient data, especially if we are collecting data from various sources/databases for comparison purposes. The easiest way for us to do this so that we automatically get the correct column names and datatypes is to use a query that joins the relevant tables but has a `WHERE` condition of `1=2`. Clearly this condition means no rows will be retrieved; however, the temporary table will be created with all the relevant meta information. And for our example, this creates a data structure with the correct names and datatypes.

Next, we call the stored procedure sp_`MSForEachDB`, with two parameters. The first parameter specifies the database to use (this is enumerated and represented by a question mark). The second parameter is the query we want to run on that database. You'll notice that we select the 10 most used indexes on each database and store their details (name and various counters) in the temporary table. Finally, we select the 10 most used indexes from the temporary table (remember this reflects all databases).

One point to note; for the results to make sense, the number of rows associated with the TOP command given with the sp_`MSForEachDB` command must match the number used for the final retrieval from the temporary table.

Summary

The purpose of this article was to explain a common pattern of code that will occur repeatedly. We discussed a quick method of creating temporary tables, and this is needed when we want to loop over all databases on the current server, extracting the results as we go.

Last updated: June 8, 2011