Dynamic SQL

by Mike Benkovich

The benefits of using a relational database management system over other types of data storage such as flat files, spreadsheets, hierarchical databases, etc., is the ability to look at the information it contains in various ways and to give us a better way of managing that content. What we learn from processing and analyzing the data leads to better insight and questions about the nature of things. For example, if we get information about annual sales volumes, we might ask which product or month is the most (or least) profitable? We might want to change the columns on the report, or sort it in a different way depending on what department we work in.

Developing applications that provide flexible paths to retrieve and manage information in large databases is one of the big challenges we face as builders of systems. Our ability to anticipate future requests and build that flexibility into the system at design time requires experience, insight, and judgment. Translating such a design into a deliverable project depends on the amount of time, the resources available, and the understanding of the technologies we are working with.

To meet this objective, most database systems provide the facilities for running SQL code directly against the database engine. ODBC has the call `SQLExecDirect`, ADO uses the `Command` object, and most others have similar calls. The purpose is to allow the developer the flexibility of creating an SQL statement within the application based on user input to determine what information to return.

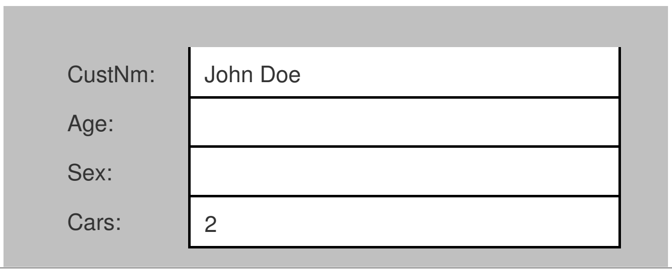
In this article, we will address the issues surrounding Dynamic SQL and its various uses. The main points of this chapter include:

- What is Dynamic SQL
- Parameterized queries
- Dynamic Scripting techniques
- Taking the next step

What is Dynamic SQL?

Dynamic SQL refers to SQL code that is generated within an application or from the system tables and then executed against the database to manipulate the data. The SQL code is not stored in the source program, but rather it is generated based on user input. This can include determining not only what objects are involved, but also the filter criteria and other qualifiers that define the set of data being acted on.

Using this approach, we can develop powerful applications that allow us to create database objects and manipulate them based on user input. For example, suppose you are working on a web application that will include a function that presents the user with a screen that contains series of parameters to define the information, then that the application performs a search based on the parameters that have been filled in.



CustNm:	John Doe
Age:	
Sex:	
Cars:	2

Without using Dynamic SQL, we would have to code the query to account for all the combinations of the various parameter fields.

```
Select * from Customer
Where ((CustNM is not null and CustNM like 'John Doe%') or CustNM is null) and
((Age is not null and Age = '') or Age is null) and
((Sex is not null and Sex = '') or Sex is null) and
((Cars is not null and Cars = 2) or Cars is null)
```

If there are only a few, it is okay, but when you have 10 or more parameters, you can end up with a complex query, particularly if you allow the user to specify conditions between parameters such as AND, OR, etc.

The more typical approach used by application developers is to use a routine that parses the fields within the client program and builds the WHERE clause to contain just the criteria needed. This results in SQL code created in the application that is based on the user input. In our applications, we can generate the query from these components to specify what we want to see and in what format.

```
Select * from Customer
Where CustName like 'John Doe%' and Cars = 2
```

Dynamic SQL vs. Stored Procedures

The purist DBA view will point out that using stored procedures would be better because of the advantages they provide. When you have the resources and time allows, encapsulating SQL into stored procedures can give us performance gains, simplify management, and result in a more secure database.

SQL Server compiles the stored procedures and saves the execution plans for future use. While we don't see the benefit with SQL that is infrequently run, a selection of something like a customer order basket, which needs to be refreshed often, will provide a noticeable performance gain. When a stored procedure is created, the optimizer will look for the best way to execute the batch of statements and determine a best approach to use, and it stores that plan for future use. Dynamic SQL is recompiled every time.

Stored procedures allow us to use parameters for input of variables and the output of results. When calling a stored procedure, we specify just the name and the values for the parameters. We don't need to send the entire query batch to the database across the network, resulting in smaller packets of information going back and forth. This reduces the conversation on the network, which in turn improves the response time to get the result set.

Because stored procedures are objects stored within the database, we can use normal **DCL** (**Data Control Language**) commands to grant and deny access. If a user has execute rights to the procedure, they assume the rights of the owner of the procedure when they call it. For example, we can create a stored procedure to return the salary of employees who work for us. By giving users rights to this procedure we can avoid granting general read access to unauthorized individuals.

Another advantage of using stored procedures is that we can encapsulate the business rules that determine the validity of the data. By creating a common routine that can be called from any application, we don't have to manage the logic in multiple places. If in our previous example, we needed to change it so that the user could only see their own salary information we could modify the stored procedure and all calling programs would use the new rule. This eliminates the need to scan the source code of all the client applications for the logic that retrieves the data and then to update it in several places. You could call this normalizing the process, just as we normalize the data structures that contain the data.

If data access is consistently implemented via stored procedures, then the sysdepends table on SQL Server will contain references to all the places that the various tables and views are used. If we need to change a table structure, we can see all the places that will be affected. Again, it is a benefit to management.

With all the advantages stored procedures have you might wonder what Dynamic SQL should be used for, and when it makes sense.

When to Use Dynamic SQL?

In the real world, we don't always have the budget or the resources to implement the perfect project. Trade offs are made and development begins before the design is completed, prototypes are added to until they are no longer prototypes but end up becoming the final application. The SQL logic necessary is not known until the development efforts are underway, and code is developed as it is needed. Sometimes the effort to coordinate between the database developers and the application teams doesn't go as smoothly as it should. Maybe the budget doesn't provide funding to pay for the database work or the staff isn't trained in writing Stored Procedures. The project is put into production and it does the job.

Other times that Dynamic SQL makes sense is for one time administrative tasks such as shrinking the database or dumping a copy of all the tables on a database. There are a number of situations that we can use the strengths of this tool to get our job done quicker and faster.

Writing Dynamic SQL

Coding effective routines that provide performance and simplification of tasks requires that we understand the intent of the tool. If misused, any tool can be a hindrance, but when applied correctly to a problem for which it was intended, Dynamics SQL really shines. In this section, we will look at how SQL Server allows us to execute Dynamic SQL, and some techniques for writing effective code.

EXEC

You have seen the EXECUTE command used to run stored procedures, but it can also be used to execute a character string. For example the simple statement to list sales by title can be called like this:

```
EXEC ('SELECT title_id, count(*) FROM sales GROUP BY title_id')
```

We are not limited to executing static strings using the EXEC command. We can generate a SQL statement based on the current environment and execute that statement as well. If we need to summarize data by the frequency of values on a particular column we could declare a local variable, set the value equal to the command we want to run. In this case we use concatenation to build the command string, and we declare a variable to hold the name of the column to group by:

```
DECLARE @col VARCHAR (50)
DECLARE @cmd VARCHAR(4000)
SET @col = 'stor_id'
SET @cmd = 'SELECT '+@col+', count(*) FROM sales GROUP BY '+@col
EXEC (@cmd)
```

This could be run from Query Analyzer as its own batch, or it could be part of a larger stored procedure. Using variables to hold names of columns or tables that may need to be changed simplifies support and maintenance of the code.

One consideration to keep in mind is that every time the database processes an EXEC command it treats the statement as a new command that needs to be treated in its own scope. This means that any variables declared within the command string being run are not visible to the calling batch, and likewise variables that are in the scope of the calling batch are not visible within the EXEC'd command.

The statement below will result in an error because the context of the variable @table is the calling batch of statements, and there is no table with the name "@table" in the database.

```
DECLARE @table VARCHAR(50)
SET @table = 'authors'
EXEC ('SELECT * FROM master..sysobjects WHERE name = @table') -- BOOM!
```

If you change the database context with the USE command the effects do not last beyond the end of the statement. This is important to keep in mind when you are working with multiple databases and don't fully qualify the tables with the database.owner.tablename syntax.

```
Use pubs
go
declare @cmd varchar (4000)
set @cmd = 'EXEC spCurrDB'
set @cmd = 'select ''The current database is: [''+d.name+'']'''
+ ' from master..sysdatabases d, master..sysprocesses p '
+ ' where p.spid = @@SPID and p.dbid = d.dbid '
EXEC (@cmd)
EXEC (N'Use master;'+@cmd)
EXEC (@cmd)
```

This example will return the name of the current database by using the @@SPID which returns the current process id and then joining the system tables sysprocesses and sysdatabases on the database id column (dbid) and then filtering the results to the row that matches our id. When you run it the first EXEC shows current context to be pubs, the second master and the third is back to pubs. When the second EXEC runs, it changes the database context just for the duration of that EXEC call, and doesn't change the calling batch's context. The database engine treats each EXEC as separate batches, which have no knowledge of the other.

If the first three characters following the EXEC statement are sp_, it assumes that you are running a system stored procedure and will search the master catalog of procedures before it looks at the current database. For that reason, it is a good idea to use a different naming standard for your own stored procedures. The performance gain might be small, but why waste resources if you don't have to?

sp_executesql

Using sp_executesql to run dynamic statements gives us a couple advantages over EXEC that are worth noting. The first is that while both evaluate the SQL statement at the point of execution, sp_executesql will store and potentially reuse execution plans while EXEC does not. The other benefit is that sp_executesql supports parameter substitution and allows you to better integrate the statements into your program.

The calling syntax for sp_executesql is as follows:

sp_executesql <@stmt> [<@param1 data_type>,<@param2 data_type>, ...]

The @stmt parameter is a Unicode string containing valid SQL commands, and the parameters are specified with a name and type. We can specify the parameters for both input and output. In this example we are going to return as output the count of books where the author is contained in the variable au_name. The output type @retType is passed as the second parameter to sp_executesql, and the variable @retVal that will be set to the returned value is passed as the third parameter.

```
declare @cmd nvarchar(4000)
declare @retType nvarchar(50)
declare @retVal nvarchar(20)
declare @au_name varchar(50)
set @@au_name = 'Ringer'
set @retType = N'@cnt varchar(20) OUTPUT'
set @cmd = N'SELECT @cnt = convert(varchar(20), count(*)) '
+ ' from titles t, titleauthor ta, authors a, sales s '
+ ' where a.au_id = ta.au_id '
+ ' and ta.title_id = t.title_id '
+ ' and s.title_id = t.title_id '
+ ' and a.au_lname like ''' + @@au_name + N''''
exec sp_executesql @cmd, @retType, @retVal OUTPUT
select @retVal
```

How we can use Dynamic SQL

In this section we will use Dynamic SQL from within single batches and stored procedures to see how it can be used in various scenarios to generate code to create tables, stored procedures and views. These examples are intended to provide you with some ideas for various approaches to situations you may need to address.

For our example we will assume that at the Northwind Company, sales are booming, and the decision has been made to partition the data into monthly sales tables. The problem is how to deal with tables and managing the data in them without changing the existing applications to account for the new schema. The goal is to minimize impact to the existing applications.

Using Dynamic SQL to create partitioned tables

The first thing we need to do is to create a procedure for creating new tables if they don't exist to hold that month's data. Dynamic SQL statements are useful in generating scripts which are dependent on the current configuration or settings. We will use an input parameter to dynamically generate the script to create a permanent table that follows a given naming standard. We create a stored procedure which takes as input parameters a date value that we will use in creating the table name, and then only create it if the table doesn't already exist. This will save us time later in that we can call this procedure without fear that we will loose data that already exists.

```
CREATE PROCEDURE spCreateSalesTable
@Create_date DATETIME = null
AS
-- If they didn't pass a date, then use the system date
if @Create_date is NULL
set @Create_date = getdate()
DECLARE @cmd NVARCHAR(255)
DECLARE @year char (2), @month char(2)
-- Next decode the date into a 2 digit year and 2 digit month
SET @year = substring(convert(VARCHAR(6), @Create_date,12),1,2)
SET @month = substring(convert(VARCHAR(6), @Create_date,12),3,2)
SET @cmd = N'CREATE TABLE Sales_' + @month + @year +
N' (stor_id char (4) NOT NULL ,
ord_num VARCHAR (20) NOT NULL ,
ord_date DATETIME NOT NULL ,
qty smallint NOT NULL ,
payterms VARCHAR (12) NOT NULL ,
title_id VARCHAR(6) NOT NULL) '
-- Only execute the create table script if it doesn't already exist
-- by checking if there is already a table in the sysobjects table
if not exists (
SELECT *
FROM dbo.sysobjects
WHERE id = object_id(N'Sales_'+@month + @year)
AND OBJECTPROPERTY(id, N'IsUserTable') = 1
)
BEGIN
exec sp_executesql @cmd
print 'Created table [Sales_'+@month+@year+']'
END
ELSE
print 'Table [Sales_'+@month+@year+'] already exists'
GO
Go
spCreateSalesTable '01-01-2003'
```

Loading Data to Partitioned Tables

Now we need to determine where to put new sales transactions. Because the table names are dependent on the sales date, we will create a stored procedure that uses dynamically generated SQL to insert the sales information into the appropriate table based on the order date. This will hide the implementation details from the application so that if we later decided we needed to change how we partitioned the sales data, we don't impact the application.

This procedure takes as input parameters the details of the sales transaction, and then uses them to generate a SQL script to perform the insert operation on the appropriate table. But before we can insert the new data, we need to ensure that the table exists, so we call the stored procedure spCreateSalesTable

```
CREATE PROCEDURE spAddSalesTran
@stor_id char(4),
@ord_num VARCHAR (20),
@ord_date DATETIME,
@qty smallint,
@payterms VARCHAR (12),
@title_id tid
AS
DECLARE @cmd NVARCHAR(255)
DECLARE @parmlist NVARCHAR(255)
declare @year char (2), @month char(2)
-- Create the sales table if it doesn't exist
EXEC spCreateSalesTable @ord_date
-- Next build the insert string
SET @year = substring(convert(VARCHAR(6), @ord_date,12),1,2)
SET @month = substring(convert(VARCHAR(6), @ord_date,12),3,2)
SELECT @Cmd = N'INSERT INTO Sales_' + @month + @year
+ ' (stor_id, Ord_num, ord_date, qty, payterms, title_id) '
+ ' values (@stor_id, @ord_num, @ord_date, @qty, @payterms,
@title_id)'
```

```
-- Setup the calling parameters for sp_ExecuteSQL
SET @parmlist = N'''+@stor_id + ''', ''' + @ord_num + ''',
'''+convert(VARCHAR(19),@ord_date)
+''', '+convert(VARCHAR(5),@qty)+', '''+@payterms+''',
'''+@title_id+''''
-- And run it
EXEC sp_ExecuteSQL @cmd, N'@stor_id CHAR(4), @ord_num VARCHAR(20),
@ord_date DATETIME,
@qty INT, @payterms VARCHAR(12), @title_id TID',
@stor_id, @ord_num, @ord_date, @qty, @payterms, @title_id
GO
```

At this point we have created a means for generating the partitioned tables and to add new records to them according to the given business rules. But querying the sales data is more complicated because we need to know the order date to get to the correct table. In the next section we will use a very useful feature of Transact SQL, namely Cursors.

Using Cursors with Dynamic SQL

You can add a lot of power to your scripts when you combine the use of cursors to drive thru system tables to generate SQL statements. In our partitioned sales table example, we may not want to expose the dynamically generated table name to the end users or force the applications to be recoded each month. We can use a view to provide a consistent view of the last 12 months of sales. Using cursors to iterate through the system tables, we can generate a script to update the view.

```
CREATE PROCEDURE spUpdateSalesView as
declare @cmd varchar (4000)
declare @Table varchar(50)
-- Create a cursor to return the tables that match our naming standard
declare myCur scroll cursor for
select name from sysobjects
where type = 'U' and name like 'Sales_%'
order by name asc
open myCur
fetch from myCur into @Table
if @@FETCH_STATUS = 0
begin
-- If we've got any data, use that first row to define the first select
set @cmd = 'create view AllSales as '
+ 'select stor_id, ord_num, title_id, ord_date, qty, payterms '
+ 'from ' + @Table
fetch next from myCur into @tABLE
while @@FETCH_STATUS = 0
begin
-- every successive table will require the UNION operator
set @cmd = @cmd + ' UNION '
+ ' select stor_id,ord_num,title_id,ord_date,qty,payterms'
+ ' from ' + @Table
fetch next from myCur into @tABLE
end
-- Next we drop the view AllSales if it exists
if exists (
select *
from dbo.sysobjects
where id = object_id(N'AllSales') )
begin
print 'Dropping View'
drop view AllSales
end
print 'Creating View'
-- And finally we add the new view definition
exec (@Cmd)
end
-- Clean up after ourselves and free up the cursor object
close myCur
deallocate myCur
GO
```

We've not got an implementation for partitioned sales tables in which we could change the implementation details without impacting the applications. Wouldn't it be nice if we had some sample data so we could test how well our solution works? In our next scenario we will add a way to do just that.

Generate Sample Data

In this stored procedure we want to provide the developer with a way to add a random sample of data spread out between two dates. We will use cursors and dynamic SQL along with the RAND operator to generate a random sampling data to load into our test database.

We pass in parameters for the number of rows we want to generate and the date range for sales records. We then use a cursor for the store and one for the titles and then using `fetch absolute` we position our cursor on a randomly determined record and use the results to generate our call to add the sales transaction.

```
CREATE PROCEDURE spCreateSampleData
@rows int, -- Number of rows to add
@mindate DATETIME, -- Minimum order date
@maxdate DATETIME -- Maximum order date
AS
-- Declare our local variables
DECLARE @store_id CHAR (4), @ord_num VARCHAR(20),
@ord_date DATETIME, @qty SMALLINT,
@payterms VARCHAR(12), @title_id VARCHAR(6),
@cnt INT, @rnd INT,
@storeCnt INT, @titleCnt INT,
@days INT, @seed INT
-- Initialize them and setup our conditions
SET nocount on
SET @cnt = 0
SET @days = datediff (day, @mindate, @maxdate)
-- Next declare a cursor to contain Store info and save the record count
DECLARE curStore scroll cursor FOR
SELECT stor_id FROM stores WHERE stor_id IS NOT NULL
OPEN curStore
SET @storeCnt = @@CURSOR_ROWS
print convert(VARCHAR(4), @StoreCnt)+ ' Stores'
```

```
-- Do the same for Titles
DECLARE curTitle scroll cursor for
SELECT title_id FROM titles WHERE title_id IS NOT NULL
OPEN curTitle
SET @titleCnt = @@CURSOR_ROWS
PRINT convert(VARCHAR(4), @titleCnt) + ' Titles'
-- Next loop until we have added the specified number of rows
WHILE @cnt < @rows begin
-- Initialize our seed value for the random number generator
SET @seed = rand() * 100000
-- Pick the store at random
SET @rnd = rand (@Seed) * @storeCnt
FETCH absolute @rnd from curStore into @store_id
-- Pick a title
SET @rnd = rand (@Seed) * @titleCnt
FETCH absolute @rnd from curTitle into @title_id
-- Pick a order date by adding a random number of days to mindate
SET @ord_date = dateadd (day, rand(@Seed) * @days, @mindate)
-- Random quantity
SET @qty = rand() * 10
-- We will use a static order number formula, and a default for terms
SET @ord_num = 'TEST'+convert(VARCHAR(5), @seed)
SET @payterms = 'Pay Terms'
-- Finally we call the stored procedure to add the sales transaction
EXEC spAddSalesTran @store_id, @ord_num, @ord_date, @qty,
@payterms, @title_id
-- Increment the counter of rows added
SET @cnt = @cnt + 1
-- Provide the user feedback by printing status update every 1000 rows
IF @cnt % 1000 = 0 -- If the remainder after dividing by 1000 = 0 then
print convert(VARCHAR(6), @cnt) + ' Rows Processed'
END
-- Now clean up after ourselves
CLOSE curTitle
DEALLOCATE curTitle
CLOSE curStore
DEALLOCATE curStore
-- update the sales view to reflect current conditions
EXEC spUpdateSalesView – Update the view of AllSales
PRINT convert(VARCHAR(5), @cnt) + ' Sales Records Generated'
GO
```

We can then use this stored procedure to generate a random set of sales data by running the following command:

```
spCreateSampleData 65000, '1-1-1999', '12-31-2003'
```

After running our data generator, we have populated a lot of information across many tables in the database. To see how the data is distributed it would be useful to be able to see what the distribution is, and whether we want to change our random data algorithms to give us a better distribution. The next example will dynamically build a SQL batch that selects the number of rows in each table, along with the name of the table.

Report on the current Table Counts

If you've ever worked with distributed databases and data replication, you have probably felt the need to be able to get some level of confidence that the data that has been loaded into a database is correct. This example provides a way to audit the row counts from every user table in the database so you can see more easily application problems caused by a table missing data in the database.

There are several approaches we could use for this problem, but in the interest of this chapter we will be using a script that generates SQL dynamically. If you wanted, you could create a stored procedure around it and pass in the name of the database to run the row counts against.

We will use the system tables again and create a cursor to drive thru the names of the user tables in the sysobjects table. If there is more than one user defined table in the database, then we will perform a UNION between queries so that we return a single rowset.

```
DECLARE @Table VARCHAR(50)
DECLARE @Cmd VARCHAR(4000)
-- Declare the Cursor to return Table objects
DECLARE myCur SCROLL CURSOR FOR
SELECT name FROM sysobjects WHERE type = 'U'
OPEN myCur
FETCH FROM myCur INTO @Table
-- If there aren't any user tables then don't do anything, otherwise...
IF @@FETCH_STATUS = 0
BEGIN
-- We set up the select statement for the first table
SET @Cmd = 'select '''+@Table+''', count(*) from '+@Table
FETCH NEXT from myCur into @Table
-- add a UNION statement in between additional tables
WHILE @@FETCH_STATUS = 0
BEGIN
SET @Cmd = @Cmd + ' UNION '
SET @Cmd = @Cmd + 'select '''+@Table+''', count(*) from '+@Table
FETCH NEXT FROM myCur into @Table
END
-- For fun, lets sort the results by the table with the most rows (col 2)
SET @Cmd = @Cmd + ' ORDER BY 2 DESC'
PRINT @Cmd
EXEC (@Cmd)
END
-- And as usual, clean up after we are done...
CLOSE myCur
DEALLOCATE myCur
```

This batch will return the table names and row counts for each so you can get the sanity check that the contents of the database are what you expect. It is helpful if you are trying to resolve production problems in which you expect all distributed copies of a read only table to have the same number of records.

Of course having the same number of rows doesn't mean that they have the same contents. If you need to audit the contents as well as the count, you need to be able to extract the data from the suspect table and compare it to the master copy. In our next example we will cover how you can extract data from a database.

Extracting a database's table contents to files

We can use the cursor approach that we used to count the rows in the tables to extract a copy of the contents to file. We will use an extended stored procedure xp_cmdshell to call the BCP (Bulk Copy Program) against each table in our cursor.

One thing to keep in mind about security and xp_cmdshell is that it spawns a command shell in the process space of the SQL Server engine itself. Therefore it will have rights to any command that the account that SQL Server runs in has. Because of this, execute rights are limited to members of the SQL Server sysadmin group, although other users can be explicitly granted this right. If the user calling xp_cmdshell is not a member of the sysadmin group then the command shell will run under the SQL Agent Proxy account and have the same rights as that account.

Because we are running BCP under the SQL Server account, using a trusted connection will give the default system admin rights. Therefore we are specifying to use a trusted connection from the BCP command line (-T). In our example, we are specifying that the output be in SQL Server native mode (-n) which means the information is written in Tabular Data Stream (TDS) the native tongue of SQL Server. We could have specified a format based extract but that would require us to also define the layout of the resulting file for each table and that would violate our objective of creating a generic script. For a complete listing of the BCP command, run it from the command line without any parameters and it will display the calling syntax.

```
DECLARE @Table VARCHAR(50)
DECLARE @Cmd VARCHAR(4000)
DECLARE myCur SCROLL CURSOR FOR
SELECT name FROM sysobjects WHERE type = 'U' ORDER BY name
OPEN myCur
FETCH FROM myCur INTO @Table
WHILE @@FETCH_STATUS = 0
BEGIN
-- setup the command line
SET @Cmd = 'bcp ' + DB_NAME() + '..' + @Table
+ ' out c:\temp\' + @table + '.dat -n -T'
-- Print the command to the screen so we can see what's happening
PRINT @cmd
EXEC master..xp_cmdshell @cmd
FETCH NEXT FROM myCur INTO @Table
END
Close myCur
DEALLOCATE myCur
```

Once the files have been created on the remote server, all we need to do is copy them to a local server and then load the data, however this will require that you already have the exact same schema defined on your local server, and that you don't have any data in the table or the duplicates will cause the process to fail. To reverse the direction and reload the data we just extracted simply change the bcp parameter out to in.

Purging Old Data

A transactional system that captures new information on a regular basis, needs to have a mechanism for keeping the databases from out-growing their hardware. Purging old data means to get rid of unneeded data records either because they are out of date, or the information has been archived to a non-transactional system for long term storage. For purposes of this example, we will create a stored procedure that will delete data between two dates. A real-life scenario would likely have other parameters and dependencies, but we will try to keep it simple for purposes of demonstration.

In the design of our stored procedure we assume that we will be passing in a date range to use. The tables that are named such that they contain data in that rage will be dropped from the database. Again, this is the blunt knife approach that would be refined in a real life scenario.

```
CREATE PROCEDURE spPurgeSales
@MinDate DATETIME,
@MaxDate DATETIME
AS
DECLARE @TblDt DATETIME
DECLARE @Table VARCHAR(50)
DECLARE @Cmd VARCHAR(4000)
DECLARE @Cnt int
-- Initialize the count so we can return how many tables were dropped
SET @Cnt = 0
-- Declare the cursor to drive thru our Sales tables
DECLARE myCur scroll cursor for
SELECT name FROM sysobjects WHERE type = 'U' AND name LIKE 'Sales_%'
OPEN myCur
FETCH FROM mycur into @Table
WHILE @@FETCH_STATUS = 0
BEGIN
-- Extract the date from the name of the table
SET @TblDt = '01-'+substring(@Table, 7, 2)+'-'+substring(@Table,9,2)
IF @MinDate <= @TblDt and @TblDt <= @MaxDate begin
-- This table matches, drop it
SET @Cmd = 'drop table ' + @Table
PRINT 'Dropping Table ['+ @Table + ']'
EXEC (@Cmd)
SET @Cnt = @Cnt + 1
END
FETCH NEXT FROM mycur INTO @Table
END
-- Clean up the cursors
CLOSE myCur
DEALLOCATE myCur
PRINT '*** ' + convert(VARCHAR(5), @Cnt) + ' TABLES DROPPED'
GO
```

System Administrative Tasks

Using SQL Server to generate system management scripts and subsequently executing them allows us to leverage the strength of SQL server to automate the processing of common administrative tasks such as checking tables using the DBCC command, extracting the database schema, and other misc. tasks.

Checking Table Integrity

SQL Server is a great tool for managing data, but it has been known to have its problems. If a page of data somehow becomes corrupted, we can identify and resolve the problem by running consistency checks against the table. The DBCC CHECKTABLE command will verify that the data pages, allocation tables and indexes are not corrupted and have reasonable data (from the database perspective of data types and values, not the application or the users). We will call it with the option REPAIR_FAST which will take care of minor problems. If SQL Server cannot fix it, it will include it as part of the result set but it is still up to us to look at these results.

```
DECLARE @table VARCHAR(50)
DECLARE @Cmd VARCHAR(4000)
DECLARE tblCur scroll CURSOR FOR
SELECT name FROM sysobjects WHERE type = 'U' ORDER BY name
OPEN tblCur
FETCH FROM tblCur INTO @table
WHILE @@FETCH_STATUS = 0
BEGIN
SET @Cmd = 'DBCC CHECKTABLE ('''+@Table+''', REPAIR_FAST)'
EXEC (@Cmd)
FETCH NEXT FROM tblCur INTO @Table
END
CLOSE tblCur
DEALLOCATE tblCur
```

Comparing Database Schemas Between Servers

As applications evolve so does the schema that they use. When deploying new versions of applications across multiple locations a common problem is ensuring that the schemas are correct and consistent. This script uses the system schema to select the names of columns and indexes for each table on the server (excluding the system tables) into a nice generic result set that can be saved to a file and then used by a comparison tool (such as WinDiff which comes with Visual Studio) to identify any differences.

```
SET NOCOUNT ON
DECLARE @dbname VARCHAR(20)
DECLARE @Cmd VARCHAR (4000)
-- First we loop thru the user defined databases on the server (dbid > 4)
DECLARE dbCur SCROLL CURSOR FOR
SELECT name FROM master..sysdatabases WHERE dbid > 4 ORDER BY NAME
OPEN dbCur
FETCH FROM dbCur into @dbname
WHILE @@FETCH_STATUS = 0 begin
-- next we build our command script
SET @Cmd = ' declare @table VARCHAR(50) '
-- Create cursor to drive thru tables in db
+ ' DECLARE tblCur SCROLL CURSOR FOR '
+ ' SELECT name FROM '+@dbName+'..sysobjects '
+ ' WHERE type = ''U'' ORDER BY NAME '
-- Open it
+ ' OPEN tblCur '
+ ' FETCH FROM tblCur INTO @Table '
-- For each table in the table cursor
+ ' WHILE @@FETCH_STATUS = 0 begin '
-- Print out the name of the current table
+ ' PRINT ''Schema ['' + @Table + '']'''
```

```
-- Then select the columns info from syscolumns & systypes
+ ' SELECT sc.name "Column Name", st.name "Type", '
+ ' sc.length "Len", sc.status "Null if 8"'
+ ' FROM '+@dbName+'..systypes st, '+@dbName+'..syscolumns sc, '
+ ' '+@dbName+'..sysobjects so '
+ ' WHERE so.name = @table and so.id = sc.id and '
+ ' sc.type *= st.type and sc.usertype *= st.usertype '
-- Next we go after the indexes, contained in sysindexes
+ ' PRINT "Indexes for [" + @Table + "]"'
+ ' SELECT si.name "Key Name"'
+ ' FROM '+@dbName+'..sysindexes si, '+@dbName+'..sysobjects so '
+ ' WHERE so.name = @table and si.id =* so.id'
-- And then we get the next table in this database
+ ' FETCH NEXT FROM tblCur into @Table '
+ ' END '
-- Clean up after ourselves
+ ' CLOSE tblCur '
+ ' DEALLOCATE tblCur '
-- @cmd is now equal to the script to get schema information
EXEC (@cmd)
-- Go on to the next database and build a new string
FETCH NEXT FROM dbCur INTO @dbname
END
CLOSE dbCur
DEALLOCATE dbCur
```

Reclaiming Unused File Space

SQL Server can automatically allocate file space as the database grows, but if you need to go the other direction, this script will loop through the system databases and run the DBCC command to shrink the files. The enterprise manager has a similar utility for shrinking the database, and it does the same basic thing, but running the enterprise manager in a highly distributed environment with hundreds or thousands of servers to work with is not efficient.

This script also uses a cursor to drive through the system tables and execute commands. I am arbitrarily using the value of 10 percent as the amount of free space to be left after the operation is complete.

```
DECLARE @dbName VARCHAR(50)
DECLARE @Cmd VARCHAR(4000)
DECLARE dbCur SCROLL CURSOR FOR
SELECT name FROM sysdatabases WHERE ID > 4 -- Exclude system databases
OPEN dbCur
FETCH FROM dbCur INTO @dbName
WHILE @@FETCH_STATUS = 0
BEGIN
SET @Cmd = 'DBCC SHRINKDATABASE (' + @dbName + ', 10)'
EXEC (@Cmd)
FETCH NEXT FROM dbCur into @dbName
END
CLOSE dbCur
DEALLOCATE dbCur
```

Going the next step

As you can see, the possibilities are endless. In the role of database administrator, using the power of dynamic SQL can make our lives much simpler. On the flip side of the coin, however, remember that that which make you can also break you. The ability to use scripts to generate scripts can be a powerful thing that can also be used to break our systems if we don't protect ourselves against it.

Access to objects in the database is checked at runtime against the rights of the user's login. Broadly granting create, update and delete rights can make you vulnerable to unintended side affects. For that reason, whenever possible, use stored procedures and grant access to them for users and applications that work with the database.

SUMMARY

In this article we covered a lot of information. We discussed the use of dynamic SQL versus stored procedures, and why you should use stored procedures when possible. We looked at how EXEC and sp_ExecuteSQL can be used to run dynamic statements at runtime. Then we went through some examples of how dynamic SQL can be used to implement a partitioned database.