# SCRIPTDOM

SUCCINCTLY®

BY JOSEPH D. BOOTH

SUCCINCTLY® EBOOK SERIES

Syncfusion®

# ScriptDOM Succinctly

**Joseph D. Booth**

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 111
Morrisville, NC 27560
USA


ISBN: 978-1-64200-246-1



**Technical Reviewer:** James McCaffrey
**Copy Editor:** Courtney Wright
**Acquisitions Coordinator:** Tres Watkins, VP of content, Syncfusion, Inc.
**Proofreader:** Graham High, content team lead, Syncfusion, Inc.

# Table of Contents

**The *Succinctly*® Series of Books ..... 9**

**About the Author ..... 10**

**Chapter 1 Introduction ..... 11**

Your day ..... 11

Overnight ..... 13

ScriptDOM ..... 14

Summary ..... 14

**Chapter 2 Getting Started ..... 15**

Create a console application ..... 15

Install ScriptDOM via NuGet ..... 15

Create the parser ..... 16

Stored procedure code ..... 17

Call the parser ..... 17

Handle errors ..... 18

Explore the tree ..... 18

Looking at the token list ..... 19

Summary ..... 19

**Chapter 3 Tokens ..... 20**

TSqlScript class ..... 20

Batches property ..... 20

Position properties ..... 20

Token properties ..... 21

TSqlParserToken class ..... 21

Using tokens ..... 21

God procedures ..... 22
Summary ..... 24
**Chapter 4 Statements ..... 25**
Stored procedure ..... 25
Create procedure statement ..... 25
Select statement example ..... 27
Summary ..... 28
**Chapter 5 Visitor Pattern ..... 29**
Visitor pattern in ScriptDOM ..... 29
Accept method ..... 29
Using the pattern with ScriptDOM ..... 30
Adding Visit methods ..... 30
Summary ..... 32
**Chapter 6 Variables ..... 33**
Variable declarations ..... 33
Visitor method ..... 34
Parameter definitions ..... 36
Variable usage ..... 37
Main code ..... 38
Summary ..... 40
**Chapter 7 Select Statement ..... 41**
Visitor method ..... 41
Determining tables ..... 43
WHERE clause ..... 47
ORDER BY clause ..... 48
GROUP BY clause ..... 49

Having clause ..... 50
Example ..... 50
Summary ..... 51
**Chapter 8 Insert Statement ..... 52**
Insert specification ..... 53
Columns ..... 53
Insert source ..... 54
Target ..... 56
SetVariableStatement ..... 56
Example ..... 57
Summary ..... 58
**Chapter 9 Update Statement ..... 59**
Target ..... 59
FROM clause ..... 60
Set clauses ..... 61
WHERE clause ..... 62
Example ..... 63
Summary ..... 63
**Chapter 10 Delete Statement ..... 64**
Target table ..... 64
WHERE clause ..... 65
Example ..... 65
Summary ..... 66
**Chapter 11 Functions ..... 67**
Identifying types of functions ..... 68
Reporting function errors ..... 69

Some other ideas....69
Summary....70
**Chapter 12 Set Commands....71**
Checking for set commands....71
SetCommandStatement....71
PredicateSetStatement....72
Other SET statements....73
SET IdentityInsert statement....73
SET RowCount statement....73
SET ErrorLevel statement....73
Summary....74
**Chapter 13 Dangerous Commands....75**
Dangerous commands....75
Takeaway....76
Summary....76
**Chapter 14 Helper Functions....77**
Helper class....77
IsTempTable()....77
ColumnType()....78
ParserForVersion()....78
SearchExpressionToText ()....79
SchemaToTableName()....80
Private methods....81
Summary....82
**Chapter 15 Generate Script....83**
SqlxxScriptGenerator (where xx is version #)....83

Format the code....84
Formatting options....85
Summary....86
**Chapter 16 SQL Versions....87**
ScriptDOM parser versions....87
Summary....87
**Chapter 17 Interacting with a Database....88**
Connecting to a server....88
Running a SQL query....89
Getting all procedure code....90
Looping through the code....91
Summary....92
**Chapter 18 Answers....93**
Newbie stored procedure....93
Report code....93
Add new company....94
User report....95
Payroll problem....96
Summary....96
**Chapter 19 Conclusion....97**
More resources....97
GitHub open source....97
Recommended links....97
General links....98
Summary....98

# The *Succinctly*® Series of Books

Daniel Jebaraj
CEO of Syncfusion®, Inc.

When we published our first *Succinctly* series book in 2012, *jQuery Succinctly*, our goal was to produce a series of concise technical books targeted at software developers working primarily on the Microsoft platform. We firmly believed then, as we do now, that most topics of interest can be translated into books that are about 100 pages in length.

We have since published over 200 books that have been downloaded millions of times. Reaching more than 2.7 million readers around the world, we have more than 70 authors who now cover a wider range of topics, such as Blazor, machine learning, and big data.

Each author is carefully chosen from a pool of talented experts who share our vision. The book before you and the others in this series are the result of our authors' tireless work. Within these pages, you will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

We are absolutely thrilled with the enthusiastic reception of our books. We believe the *Succinctly* series is the largest library of free technical books being actively published today. Truly exciting!

Our goal is to keep the information free and easily available so that anyone with a computing device and internet access can obtain concise information and benefit from it. The books will always be free. Any updates we publish will also be free.

**Let us know what you think**

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctlyseries@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on social media and help us spread the word about the *Succinctly* series!




# About the Author

Joseph D. Booth has been programming since 1981 in a variety of languages, including BASIC, Clipper, FoxPro, Delphi, Classic ASP, Visual Basic, Visual C#, and the .NET Framework. He has also worked in various database platforms, including mySQL, PostgreSQL, Oracle, and SQL Server.

He is the author of 11 Syncfusion Succinctly titles including *Accounting Succinctly*, *Regular Expressions Succinctly*, and *SQL Server Metadata Succinctly*, as well as six books on Clipper and FoxPro programming, network programming, and client/server development with Delphi. His latest project is a SQL analysis tool called SQL Sleuth.

Joe has worked for a number of companies including Sperry Univac, MCI-WorldCom, Ronin, Harris Interactive, Thomas Jefferson University, People Metrics, and Investor Force. He is one of the primary authors of Results for Research (market research software), PEPSys (industrial distribution software), and a key contributor to AccuBuild (accounting software for the construction industry).

He has a background in accounting, having worked as a controller for several years in the industrial distribution field, but his real passion is computer programming. In his spare time, Joe is an avid tennis player, practices yoga and martial arts, and spends as much time as possible with his grandkids, Blaire and Kaia.

# Chapter 1 Introduction

Imagine this: you just started your new job as a database administrator. It's at a great company with quite a few SQL developers and a large number of databases. It's early morning, one cup of coffee in, when your first email chimes its arrival in your inbox.

## Your day

The email is from Bradley, one of the newest SQL developers, and he sends you a copy of the stored procedure he is working on. He asks, "What is wrong with the stored procedure? It doesn't work properly." So you pull up the code, which is shown in Code Listing 1.

*Code Listing 1: Newbie stored procedure*

```
CREATE procedure [FindUsersByLanguage](@languagecode varchar)
as
begin
      set nocount on
      declare @numPeopleFound int
      select * from [Person]
      where LanguageID is null or LanguageID <> @languagecode
end
```

You send off a quick reply and hope Bradley learns from this and fixes it quickly. A few minutes later, Lauren, a project manager, calls and complains about the performance of a particular report. Figure 1 shows the table, and the code is in Code Listing 2.

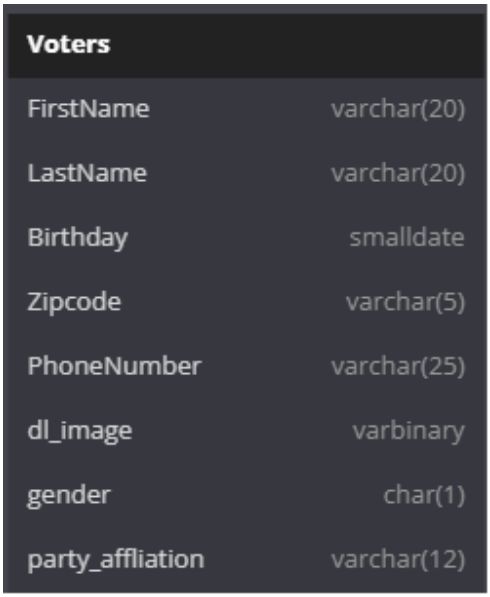


*Figure 1: Voters table*

*Code Listing 2: Report code*

```
CREATE procedure [dbo].[PhoneList] (@whichparty varchar(12))
as
begin
      select * from dbo.voters
      where upper(party_affiliation) = @whichparty
end
```

You ask Lauren for the developer's name (Kaitlin) and quickly send her an email telling her how to fix the code.

There are a few more fires to put out in the morning, but so far, so good. Of course, the first email after lunch is a production issue. Glen says the following code has been working for quite some time, but now, it's not working as expected:

*Code Listing 3: Add new company*

```
CREATE procedure [dbo].[InsertCompany] (@companyName varchar(50),@zipCode
varchar(10) )
as
begin
      DECLARE @companyID int        -- Record # of new company

      INSERT INTO dbo.Company (CompanyName,PostalCode)
      VALUES (@companyName,@zipcode)

      SET @companyID = @@IDENTITY

      UPDATE dbo.Company
      SET city = zp.City,StateCode = zp.StateCode
            FROM (select * from dbo.ZipCodes WHERE zipCode=@zipCode) zp
      WHERE CompanyID = @companyID
end
```

Hmm, this is a bit unexpected, but you do see the potential issue. You confirm your suspicions and send a quick reply to Glen.

It's almost the end of the day when the next call comes in: there is another very slow-running report. There is a table (**ASP Net Users**) containing every user (about 110,000 rows) across all stores. Each store has its own list of users (**Store Users**), typically 800–1,000 users.

Running this query takes over a minute to return 850 rows. Code Listing 4 shows the code that is running slowly.

*Code Listing 4: User report*

```
CREATE procedure [dbo].[UsersReport]
as
```

```
begin
      SELECT      sur.Email
                  ,u.UserName as 'UserName'
                  ,u.FirstName as 'First Name'
                  ,u.LastName as 'Last Name'
      FROM [StoreUser] sur
      JOIN [AspNetUsers] u ON u.Id = sur.id
End
```

Figure 2 shows the two tables referenced in the query.

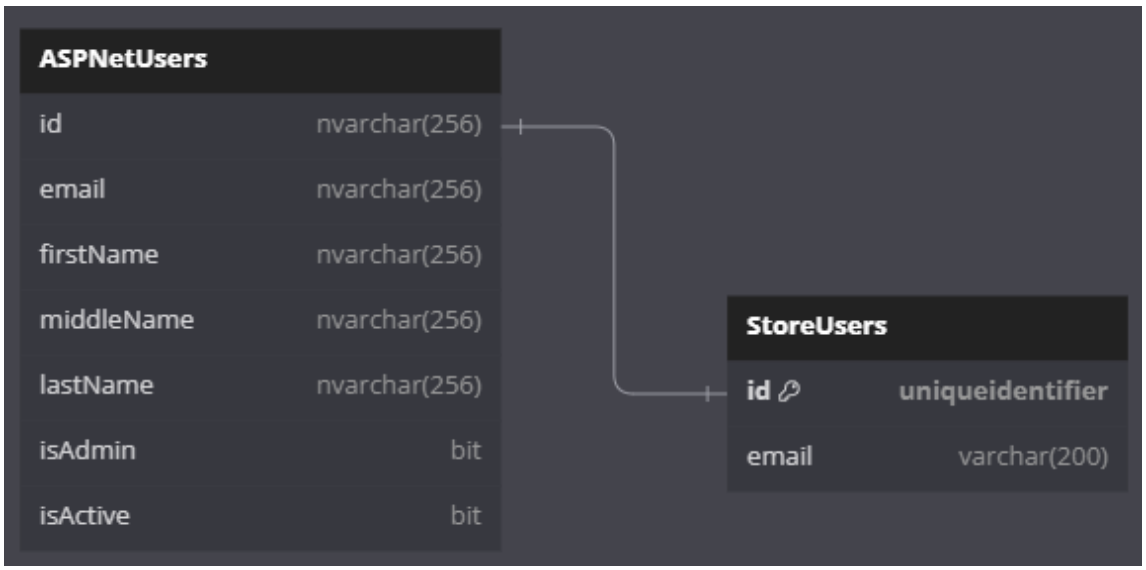


*Figure 2: User tables*

It's a bit tricky, but you see the error and show the developer, Christine, a quick fix. At first, she chuckles when you fix it with one change—but she is surprised and impressed when the report now runs in one second.

It's been a long day, but now it's finally time to go home!

## Overnight

After a long day of looking at code, you are ready to head home. What have you gotten into? Wouldn't it be nice to make sure all the code in the database can be checked? That is the role of a SQL lint checker, and there are many SQL lint checkers available. However, Microsoft has a hidden gem that allows you to roll your own custom checker that takes into account issues that are unique to your system: ScriptDOM, a .NET library.

## ScriptDOM

ScriptDOM is short for T-SQL Script Document Object Model. ScriptDOM parses SQL code and produces a list of tokens and an abstract syntax tree (AST). You can walk through the tokens list, navigate the AST, or both. ScriptDOM also uses the visitor pattern (discussed in Chapter 5), which allows you to find just the statements or elements you are interested in.

> ***Note: ScriptDOM was originally created in 2006 as an internal tool at Microsoft. It was first released to the public with SQL 2008, and in April 2023, the source code was made available in this GitHub repository.***

ScriptDOM performs the three basic parser steps that most compilers and natural language processing tools use:

- The **lexer** (lexical analysis step) breaks the script into a list of tokens.
- The **parser** reads the token lists and builds the abstract syntax tree.
- The **syntax checker** confirms that the AST is valid and will either report that the script contains errors and report these errors, or accept it as a valid syntax for the SQL code.

You can access the token list and the abstract tree if you want to parse the procedure directly. The visitor pattern allows you to hook into the process and get statements you are interested in. In a recent project I was involved with, we needed to check all `String_Split` functions to see if we were using the user-defined function (UDF) a developer created or the SQL-native version. We can hook a visitor pattern to return all function calls it encounters and provide details about which version of this particular function is being used. (See Chapter 11 for more about dealing with functions.)

You can use the ScriptDOM library to:

- Check a SQL script for errors.
- Format SQL code.
- Look for problems.
- Check for company standards.

ScriptDOM has been in the .NET Framework for quite some time, since SQL 2008, and was recently released as open source.

## Summary

In this ebook, we are going to look at ScriptDOM and how to install and use it. Then, we will create some code to identify each of the issues described in this chapter. This library has been part of the .NET Framework for quite some time, and it's one of the many hidden gems buried in .NET.

> ***Note: If you are curious, the answers to all of the procedures in this chapter are summarized in Chapter 18.***

# Chapter 2  Getting Started

In this chapter, we will set up a simple console application to call the ScriptDOM library and let it parse a basic stored procedure. This will provide a starting point for interacting with the library.

## Create a console application

I am using Visual Studio 2022 and creating an empty console application. It should look something like Code Listing 5, an empty shell program.

*Code Listing 5: Empty console program*

```
namespace Chap2;

internal class Program
{
    private static void Main(string[] args)
    {
    }
}
```

## Install ScriptDOM via NuGet

Search for the ScriptDOM library in NuGet and then install it in your solution. Figure 3 shows the NuGet browser. Once you find the package, go ahead and install it in your project.

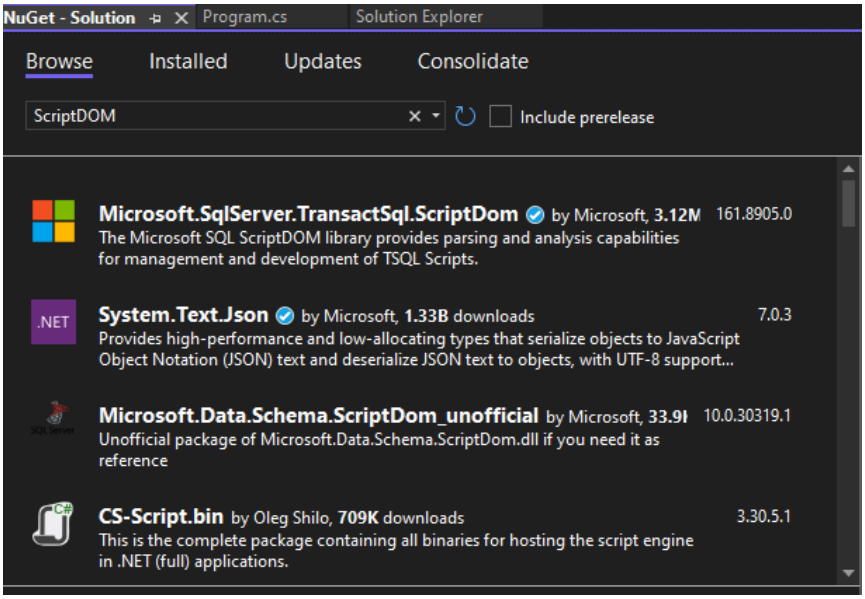


*Figure 3: NuGet for ScriptDOM*

## Create the parser

Once you have installed the NuGet package, you can create the parser and supporting variables. Add the code in Code Listing 6 to your empty **program** file.

*Code Listing 6: Create parser object*

```
        Console.WriteLine("ScriptDOM Parser - Hello, World!");
        Console.WriteLine("");

        // Create a reference to the ScriptDOM library
        // Note that 160 refers to SQL version 2022
        var parser = new TSql160Parser(true, SqlEngineType.All);

        // Create a list to hold any errors
        IList<ParseError>? errors = null;
```

The **`TSql160Parser`** is a ScriptDOM parser that handles SQL 2022 (see Chapter 15 for parser versions). You also need a list to hold any errors that might occur while attempting to parse the SQL code. Be sure to add the **`using`** statement to your program. The SQL engine enum type parameter can be **`All`**, **`SqlAzure`**, or **`Standalone`**.

```
using Microsoft.SqlServer.TransactSql.ScriptDom;
```

## Stored procedure code

For this basic example, you are going to create a stored procedure directly in the code. Code Listing 7 shows a sample stored proc.

*Code Listing 7: Sample stored proc*

```
        // For simplicity, generate a stored procedure
        StringBuilder sb = new StringBuilder();

        sb.AppendLine("--  Chap 2 Sample procedure");
        sb.AppendLine("");
        sb.AppendLine("CREATE PROCEDURE dbo.GetNewCustomers");
        sb.AppendLine("AS ");
        sb.AppendLine("BEGIN");
        sb.AppendLine("     DECLARE @30DaysAgo Date = DATEADD(day,-
30,getDate() )");
        sb.AppendLine("     SELECT * FROM customers");
        sb.AppendLine("     WHERE dateAdded >= @30DaysAgo");
        sb.AppendLine("END");
        sb.AppendLine("GO");
        sb.AppendLine("EXEC dbo.GetNewCustomers");
```

You also need to convert the string into a stream for the parser, as shown in Code Listing 8. This will be a standard code snippet you will need to interact with ScriptDOM.

*Code Listing 8: Convert to a stream*

```
        // Copy string (from StringBuilder) to a stream
        byte[] byteArray = Encoding.ASCII.GetBytes(sb.ToString());
        MemoryStream stream = new MemoryStream(byteArray);
        StreamReader rdr = new StreamReader(stream);
```

## Call the parser

The next step is to call the parser itself, passing the created stream. Code Listing 9 shows the code to call the parser directly. It takes two parameters: the stream containing the procedure code and an output parameter to return the list of any parsing errors.

*Code Listing 9: Call the SQL parser*

```
        // Ask ScriptDOM to parse it
        TSqlScript tree = (TSqlScript)parser.Parse(rdr, out errors);
```

***Note: Setting the type, rather than just `var`, is important throughout ScriptDOM, because some of the base general classes will not have access to various properties. For example, if `tree` in Code Listing 9 was `var`, the `Batches` collection would not be accessible.***

## Handle errors

If the parser detects errors because the code fails, the **`errors`** list out parameter will contain all of the errors the parser encountered. Code Listing 10 shows code to display the errors to the console.

*Code Listing 10: Display any errors*

```
        // If parsing fails, show the error messages
        if (errors.Count > 0)
        {
            foreach (ParseError err in errors)
            {
                Console.WriteLine(err.Message);
            }
        }
```

These errors would be the same errors you would get in SSMS (SQL Server Management Studio). If you were to remove the **`AS`** line from the code, for example, you would encounter an error message like **`Incorrect syntax near BEGIN`**.

## Explore the tree

Assuming no errors are found, ScriptDOM will generate a token list and an abstract syntax tree that contains a list of the statements and clauses for the code. Code Listing 11 loops through the token list and displays the results to the console.

*Code Listing 11: Show the token list*

```
            // Show the token list
            for(int x=0;x<tree.ScriptTokenStream.Count-1;x++)
            {
                TSqlParserToken token = tree.ScriptTokenStream[x];
                Console.WriteLine(  token.Line.ToString()+
                              "  "+token.TokenType.ToString()+": "+
                                   token.Text.ToString());
            }
```

The complete code can be downloaded from GitHub.

## Looking at the token list

Based on our stored procedure, let's look at a single line of code and see the token stream it generated. The **`CREATE PROCEDURE`** line follows:

```
CREATE PROCEDURE dbo.GetNewCustomers
```

This will return a token list, as shown in Figure 4.

```
3  Create: CREATE
3  WhiteSpace:
3  Procedure: PROCEDURE
3  WhiteSpace:
3  Identifier: dbo
3  Dot: .
3  Identifier: GetNewCustomers
3  WhiteSpace:
```

*Figure 4: Token list from CREATE PROCEDURE*

You might, for example, want to require that all create procedure and alter procedure commands specify a schema name. If only one identifier is found, that would suggest a possible error: that a schema name was not provided.

## Summary

In the next chapter, we will describe the abstract tree and token list returned from the parser. We will play around a bit by modifying the sample procedure to generate errors (for example, removing the **`AS`** statement) or to view the various tokens. Although the token list is useful, when combined with the visitor pattern (Chapter 5), you can see just how powerful a tool ScriptDOM is.

# Chapter 3 Tokens

The abstract syntax tree returned by the **parser** call contains a list of statements from the SQL code. In this chapter, we will explore the script and statement objects and the information they contain.

## TSqlScript class

The tree object returned from the **Parse** method is a type **TSqlScript**. The following properties are defined for the class:

### Batches property

If more than one script appears (separated by the **GO** command), each script will have its own **TSqlBatch** object. Table 1 shows the properties of the **TSqlBatch** class.

*Table 1: TSqlBatch class*

| Property | Type | Description |
|---|---|---|
| **FirstTokenIndex** | Int | First element # in the list of tokens found in the script |
| **FragmentLength** | Int | Length of the text from the script |
| **LastTokenIndex** | Int | The last element # in the list of tokens |
| **ScriptTokenStream** | List | Collection of parser tokens |
| **StartColumn** | Int | Column position in the source file |
| **StartLine** | Int | Line in the source file |
| **StartOffset** | Int | Offset (byte #) in the original source file |
| **Statements** | List | List of statement types (Create procedure, select, etc.) |

The **Statements** list will contain a variety of different statements. This will be a nested tree of the type of SQL statements it finds. We will explore this in detail in Chapter 4.

### Position properties

The **StartOffset** and **FragmentLength** integer properties provide the offset and size in the script of the specific element. The **StartLine** and **StartColumn** integer properties provide similar information for the line and column number in the script.

## Token properties

The **FirstTokenIndex** and **LastTokenIndex** integer properties provide the starting and ending element numbers in the **ScriptTokenStream** collection. The **ScriptTokenStream** property is a list of **TSqlParserToken** objects derived from the script.

# TSqlParserToken class

The **ScriptTokenStream** list contains a collection of all parser tokens found in the script. Each element is a **TSqlParserToken** object, which contains the properties listed in Table 2.

*Table 2: TSqlParserToken properties*

| Property | Type | Description |
|---|---|---|
| **Column** | Int | Column position in the source file |
| **Line** | Int | Line in the source file |
| **Offset** | Int | Offset (byte #) in the original source file |
| **Text** | String | Text content of the token |
| **TokenType** | TSql Token Type | Enumerated token type; for example:<br>• Single line comment<br>• White space, text contains space, CR, LF, etc.<br>• Identifier (schema, procedure name, etc.)<br>• Variable: a variable or parameter<br>• Declare: declare variable statement<br>• Various statements and fragments:<br>  ○ **SELECT**, **FROM**, **WHERE**, etc. |

The list of tokens is arranged as found in the script. There is no implied relationship between the list; it is simply a sequential list of the tokens from the code. You can see the complete list of token types at TSqlTokenType Enum.

# Using tokens

Although you could process the token list for individual code snippets, the statements and visitor pattern (see later chapters) are better options for processing code statements. Tokens can be used for aggregate data about the code, such as the number of variables or the percentage of comments.

## God procedures

A *God procedure* (or code) is a routine that is generally doing too much. In programming, it is usually better to write small code modules rather than a single monolithic code component. Smaller modules are easier to debug and much more likely to be reusable. We might want to see a couple of things to check for to determine if our procedure might be too big. How big is it? How many tables?

Listing 12 shows a sample stored procedure to compute payroll. We are going to use the token list to look for certain parameters that might suggest a code review is needed.

*Code Listing 12: Compute payroll*

```
--  Chap 3 Sample procedure
/*
Purpose: Compute payroll amount, update employee records,
         email data file to vendor, process result from vendor
   Date: 02-FEB-2021
*/
CREATE PROCEDURE dbo.ComputeEntirePayroll(@PayrollDate date)
AS
BEGIN
     DECLARE @15DaysAgo Date = DATEADD(day,-15,@payrollDate )
     DECLARE @PayAmount MONEY;

     SELECT em.Id,em.HourlyRate*ts.HoursWorked as PayAmount,em.SSN
     INTO #tmpPayroll
     FROM employeeMaster em
     JOIN timesheets  ts on ts.employeeId =  em.id
     WHERE ts.workdate between @15DaysAgo and @payrollDate

     -- Flag timesheet as processed
     UPDATE timesheets SET payDate =@payrollDate
     WHERE workdate between @15DaysAgo and @payrollDate

     -- Update employee and send payroll data to vendor
    DECLARE payCursor CURSOR
       FOR SELECT * FROM #tmpPayroll;

    OPEN payCursor
    -- Code to use cursor...
    CLOSE payCursor
    DEALLOCATE payCursor
END
```

You could visually look at the code and probably decide whether to review it or not, but with a large number of procedures, it could be helpful to have some automated ways to suggest which ones to review.

***Tip: For a quick test, see if you can determine the bug that causes people to be overpaid. See Chapter 18 if you want the answer.***

## Analyzing the code

We are going to take our tree (returned from the **Parse** method) and compute some summary values about the code.

*Code Listing 13: Create the token list*

```
        // Ask ScriptDOM to parse it
        TSqlScript tree = (TSqlScript)parser.Parse(rdr, out errors);
```

Once we have the **tree** object, we can use LINQ (Language Integrated Query) to collect some statistics about the code. Listing 14 illustrates some queries.

*Code Listing 14: LINQ to get info*

```
// Gather information about the code
var sm = tree.ScriptTokenStream;
int numberComments = sm.Count((cc) =>
                     cc.TokenType ==  TSqlTokenType.SingleLineComment ||
                     cc.TokenType == TSqlTokenType.MultilineComment);
int numberSelect = sm.Count((cc) => cc.TokenType == TSqlTokenType.Select);
bool anyCursors = sm.Any((cc) => cc.TokenType == TSqlTokenType.Cursor);

// Since the table is an identifier (which can be multiple items),
// we simply count FROM and JOIN to get table references)
int numberTables = sm.Count((cc) =>
                    cc.TokenType == TSqlTokenType.From ||
                    cc.TokenType == TSqlTokenType.Join);
// Compute percent of comments over entire script length
double commentsRatio = sm.Where((cc) =>
                   cc.TokenType == TSqlTokenType.SingleLineComment ||
                    cc.TokenType == TSqlTokenType.MultilineComment).Sum((cc)
                   => cc.Text.Length) * 1.0 / tree.FragmentLength;

// Number of statements that manipulate data
int numberUpdateStatements = sm.Count((cc) =>
                   cc.TokenType == TSqlTokenType.Insert ||
                    cc.TokenType == TSqlTokenType.Delete ||
                    cc.TokenType == TSqlTokenType.Update);
```

***Note: You can download the LINQ Succinctly ebook by Jason Roberts from the Syncfusion website if you want to learn more about LINQ.***

With these values, we can now make a quick decision about whether we should review the code. Figure 5 shows sample console output from the analysis of the stored procedure.

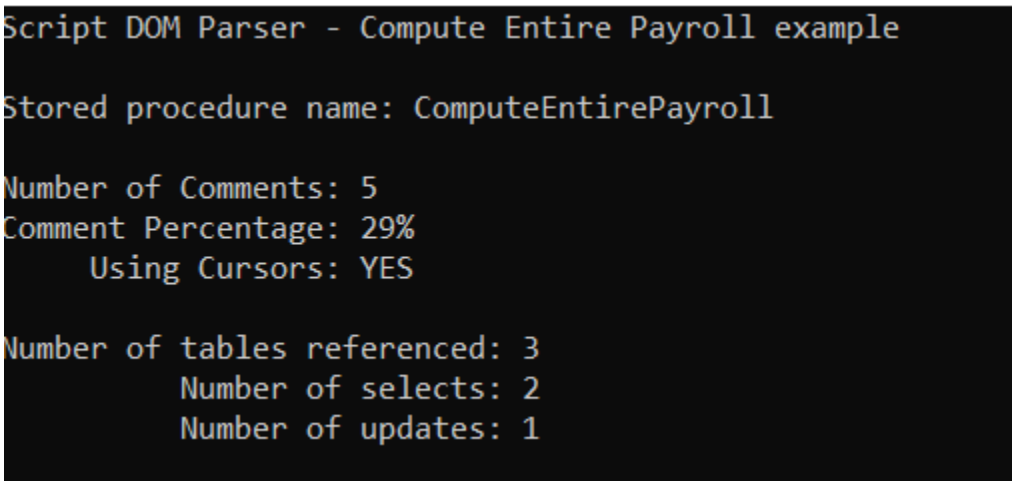


*Figure 5: Stored proc analysis*

You can use the various values to determine if the procedure might be too large or needs a review. In this example, issues like cursors, multiple **`select`** statements, and low comment percentage might suggest a code review.

## Summary

The **`Parse`** method provides both a sequential list of tokens (**`Script Token Stream`**) and a list of statements (**`Statements`** property in the batches). Tokens in a list collection can be very useful for getting basic information about the code, such as how big it is and how many comments it has.

You can write code to explore the statements collection as part of any script analysis (see Chapter 4), and the visitor pattern (Chapter 5), which makes processing individual statements easier.

# Chapter 4 Statements

The **Statements** property is a nested collection of the various statements within the script. Each statement type will have its own set of properties applicable to that statement. Looking at the batches collection from our example code in Chapter 3, we will find a property called **statements**, which is a collection of all statements. A statement is a SQL command or the content of a **BEGIN…END** block. In our example, our first statement is the **Create Procedure** statement.

## Stored procedure

```
CREATE PROCEDURE dbo.GetNewCustomers(@StartDate date)
AS
    BEGIN
        DECLARE @30DaysAgo Date = DATEADD(day,-30,getDate() ),
                @NumDays int = 30;
        SELECT * FROM customers
        WHERE dateAdded >= @30DaysAgo OR dateAdded>=@StartDate
    END;
```

*Figure 6: Simple stored proc*

## Create procedure statement

Some of the properties of the **CreateProcedure** statement are shown in Table 3.

*Table 3: CreateProcedure statement class*

| Property | Type | Description |
|---|---|---|
| **Parameters** | List | An array of parameter objects from the statement |
| **ProcedureReference** | Procedure reference object | This object has details of the procedure, including a name, and the identifiers (schema and name) |
| **IsForReplication** | Boolean | The **Replication** option value |
| **StatementList** | List | A collection of child statement objects |

The **CreateProcedure** statement contains detailed information, but also a list of the statements within the block (i.e. between the **CREATE** statement and the end of the procedure). The **StatementList** property is a collection of all statements within the block.

The first statement in the **Statements** list is the **BeginEndBlock** statement, as shown in Figure 7.

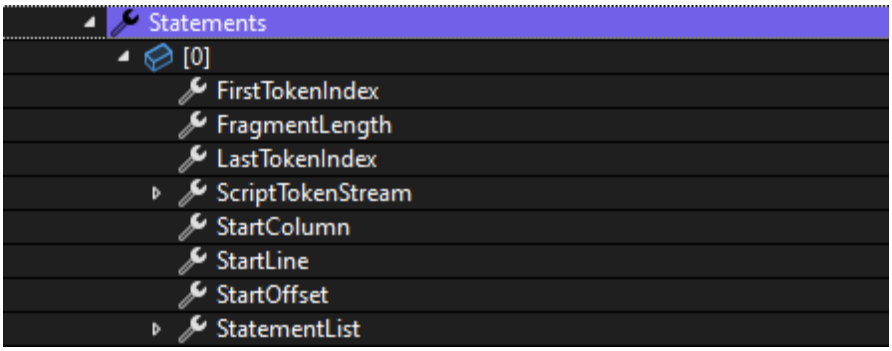


*Figure 7: BeginEndBlock Statement*

This block includes all statements within the **BEGIN…END** block in the stored proc. We can now open the **StatementList** property, which will show a collection of two statements—all the statements within the **BEGIN…END** block:

- **DeclareVariableStatement**
- **SelectStatement**

These are the lowest-level statements in the tree and they will not have any **Statement List** property associated with them.

Tracing this down, you will see a tree of all the statements that we found with the script. Figure 8 shows the statements.

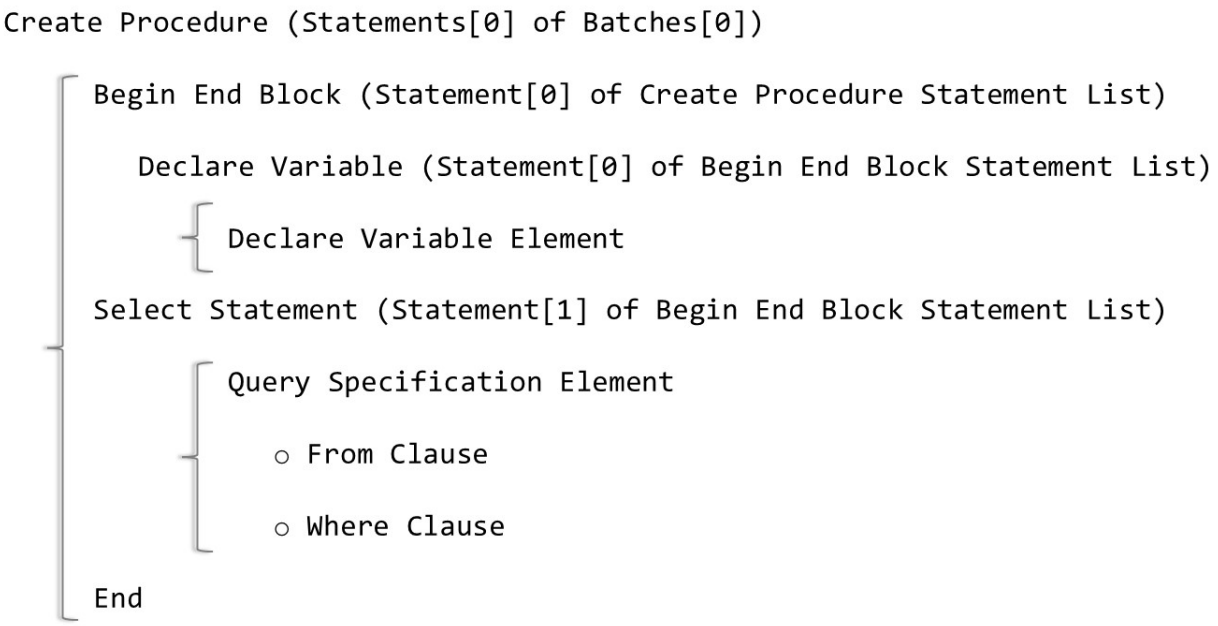


*Figure 8: Nested statements*

Each statement in the list has its own class definition, with appropriate properties based on the statement type. For example, the **`FROM`** clause has a property called **`TableReferences`**, a list of all tables within the query. The **`WHERE`** clause has a property called **`SearchCondition`**, which provides the expression, comparison method, and so on.

While you can work directly with these various statement classes, the ScriptDOM library relies on the visitor class pattern (described in Chapter 5), which provides a much cleaner approach than parsing your way through the statement list.

## Select statement example

In our example, once we get to the **`SELECT`** statement, we can access properties appropriate to interpret a **`SELECT`** statement. Table 4 shows a partial list.

*Table 4: SELECT statement properties (partial)*

| Property | Content | Description |
|---|---|---|
| **`Into`** | Null | There is no **`INTO`** clause in the **`SELECT`** |
| **`QueryExpression`** | customers | Contains a **`FROM`** clause, which has a table reference with a value |
| **`GroupBy`** | Null | No **`GROUP BY`** in the **`SELECT`** |
| **`OrderBy`** | Null | No **`ORDER BY`** in the **`SELECT`** |

| Property | Content | Description |
|---|---|---|
| **WHERE** clause | **dateAdded**<br>**>=**<br>**@30daysAgp** | **SearchCondition** object that contains the first expression (**dateAdded**), the comparison type (**GreaterThanOrEqualTo**), and the second expression (**@30DaysAgo**) |

Although it requires some familiarity with the nested objects, the **Statement** object provides all the parsing information necessary to interpret the statement. In subsequent chapters, we will look at the various statements in more detail.

# Summary

The **Statement** list can be tricky to navigate, as nested statements can get pretty deep in the list of statements. However, once you've found the statement you want to process, the data type of that statement will provide the necessary details to interpret and analyze the statement. The next chapter provides another approach for getting statements from the parser tree.

# Chapter 5 Visitor Pattern

The visitor pattern is a software design pattern from the group of behavioral patterns. It is one of 23 “Gang of Four” design patterns from the book *Design Patterns*, written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

It allows you to add new methods (behaviors) to an existing class without modifying the original class. The **`TSqlScript`** class from ScriptDOM has a collection of **`Statement`** objects, such as the **`Select`** and **`Declaration`** statements. By creating a **`visitor`** class, you can create a separate method to perform operations on the different statements within the **`TSqlScript`** object.

***Tip: If the visitor pattern is new to you, think of it like an event-driven element in Windows, but without a UI portion. For a button, your code can be called when an event, such as a click, occurs. The visitor pattern is similar: you define code to be called when the parser encounters a particular kind of statement.***

## Visitor pattern in ScriptDOM

When writing any analysis code, we will want to perform unique operations, depending upon the type of statement. This pattern allows us to create a class containing methods to handle various statements. The statement will be sent to the appropriate method based on the statement type parameter.

## Accept method

The **`TSqlScript`** class has a method called **`Accept()`**. You will need to create an instance of your **`visitor`** class and use it as a parameter to the **`Accept`** method call. For example, Code Listing 15 shows a code snippet to create the **`visitor`** class and pass it to the **`Accept`** method.

*Code Listing 15: Creating and accepting a visitor object*

```
TSqlScript tree = (TSqlScript)parser.Parse(rdr, out errors);
    Visitor treeData = new();
    tree.Accept(treeData);
```

***Note: The `new()` without the class name is a feature in Visual Studio 2022. If you are using earlier versions, you will need to include the class name after `new()`.***

In this example, we are creating and saving the **`Visitor`** object in case we need the properties after the parsing has completed. You can also create the visitor directly on the accept call, but then you will not have access to any visitor properties after the parsing has completed.

```
            tree.Accept( new Visitor() )
```

## Using the pattern with ScriptDOM

To use the pattern, you need to create a **`Visitor`** class that can be used as a parameter to the **`Accept()`** method. Listing 16 shows a shell for the **`Visitor`** class.

*Code Listing 16: Visitor class*

```
    public class Visitor : TSqlFragmentVisitor
    {
        public Visitor() {
        }
        // Visitor statements
        public override void ExplicitVisit(DeclareVariableStatement node)
        {
            base.ExplicitVisit(node);
        }
    }
```

The constructor is not required but can be added if you'd like to do any initialization prior to the visitor methods being called. The **`ExplicitVisit`** method is overridden for each statement you would like to process (have the **`TSqlScript`** object **`Visit`**). The final statement in your **`override`** method should call the **`base`** method and pass the parameter along.

> ***Tip: Visitor methods also accept a `Visit()` method. If you call a `Visit` method, but then call the `ExplictVisit` from the base (or vice versa), the code will compile, but will not work. In addition, if you forget to call the `base` method, any visitor methods after that call will not work.***

### Adding Visit methods

You can add individual methods for the statements you want to process. Using our initial code example, we might want to process the **`DECLARE`** statement and the **`SELECT`** statement. Listing 17 shows those **`Visit()`** methods.

*Code Listing 17: Visitor methods*

```
        // Visitor statements
        public override void ExplicitVisit(DeclareVariableStatement node)
```

```
        {
            base.ExplicitVisit(node);
        }
        public override void ExplicitVisit(SelectStatement node)
        {
            base.ExplicitVisit(node);
        }
```

All of the possible statements are available as parameters to the **`ExplicitVisit`** methods. Table 5 shows some common statement parameters.

*Table 5: Some useful statement parameters*

| Parameter Name | Description |
|---|---|
| **`DeclareVariableStatement`** | Variable declarations |
| **`VariableReference`** | Each time a declared variable is used |
| **`SelectStatement`** | Each **`SELECT`** statement in the script |
| **`DeleteStatement`** | **`DELETE`** statements |
| **`UpdateStatement`** | **`UPDATE`** statements |
| **`CreateProcedureStatement`** | Creating a new stored procedure |
| **`SetCommandStatement`** | Code to set SQL options |

Each statement type will have its own set of properties appropriate for the statement. In addition to the unique properties, the statement classes have a set of common properties. Table 6 lists the common statement properties.

*Table 6: Common statement class properties*

| Property | Type | Description |
|---|---|---|
| **`FirstTokenIndex`** | Int | First element # in the list of tokens found in the script |
| **`FragmentLength`** | Int | Length of the text from the script |
| **`LastTokenIndex`** | Int | The last element # in the list of tokens |
| **`ScriptTokenStream`** | List | Reference to the collection of parser tokens |
| **`StartColumn`** | Int | Column position in the source file |
| **`StartLine`** | Int | Line in the source file |
| **`StartOffset`** | Int | Offset (byte #) in the original source file |

You can use these properties to return line and column numbers, or perhaps to visibly highlight the issue in a source window.

## Summary

The visitor pattern allows you to cleanly process any statement from the stored proc, without relying on conditional testing. We allow the **`TSqlScript`** class to handle the statement routing, and we only need to write code for whichever statements we want to analyze further.

# Chapter 6 Variables

In this chapter, we will use ScriptDOM to get a list of variables and parameters from a stored procedure. Let's consider our newbie script from Chapter 1. Code Listing 18 shows that procedure.

*Code Listing 18: Newbie procedure*

```
CREATE procedure [FindUsersByLanguage](@languagecode varchar)
as
begin
      set nocount on
      declare @numPeopleFound int
      select * from [Person]
      where LanguageID = null or LanguageID <> @languagecode
end
```

Some problems include:

- No size specified on the language code parameters
- Unused variables
- No schema specified on the create procedure

In this chapter, we will use ScriptDOM to find each of these issues.

## Variable declarations

The **`DeclareVariableStatement`** exists for each **`declare`** statement and the list of variables. In addition to the standard statement properties defined in Chapter 4, this statement also has a **`Declarations`** property, which is a list of all declaration variable elements found in the script.

The key properties of the **`DeclareVariableElement`** class are shown in Table 7.

*Table 7: DeclarationVariableElement key properties*

| Property | ScriptDOM class | Description |
|---|---|---|
| **`DataType`** | SQL data type reference | **`SQLDataType`** option that contains the data type name (**`date`**, **`int`**, etc.). See SQL Data Type Enum to see the data types. |
| **`Value`** | Function call | If value is set from a function, function details can be derived from the **`FunctionCall`** class |
| | (Integer, float, etc.) Literal | The nested **`Value`** property will contain the value of the literal data type |

| Property | ScriptDOM class | Description |
|---|---|---|
| **VariableName** | Identifier | This class contains the name of the variable |

## Visitor method

The visitor method call to get all **Declare** statements is shown in Code Listing 19.

*Code Listing 19: Visit method for declared variables*

```
        public override void ExplicitVisit(DeclareVariableStatement node)
        {
            base.ExplicitVisit(node);
        }
```

Within the method code, we can process the **Declarations** list. We will extract the variable name and the line and row number. We also have a string indicating a variable (**V**) or parameter (**P**). Also, we add an empty slot to hold the number of times the variable is referenced. Listing 20 shows the code to load the declarations into a dictionary object.

*Code Listing 20: Visitor class*

```
namespace Chap6
{
        public class Visitor : TSqlFragmentVisitor
        {
            private Dictionary<string, varInfo> variables =
                                new Dictionary<string, varInfo>();

            public struct varInfo
            {
                public int LineNumber;
                public int ColumnNumber;
                public int TimesVariableUsed;
                public string VarType;  // V=variable, P=parameter
                public string DataType;
                public int DataSize;
            }
            // Class properties
            public Dictionary<string, varInfo> Variables
            { get => variables; set => variables = value; }

            public string ProcedureName;
            public string SchemaName;

            public Visitor()
            {
```

```
            }
            // Visitor statements
            /// <summary>
            /// Get any variable declarations
            /// </summary>
            /// <param name="node"></param>
            public override void ExplicitVisit(DeclareVariableStatement
node)
            {
                foreach (DeclareVariableElement varObj in
node.Declarations)
                {
                    SqlDataTypeReference dt =
(SqlDataTypeReference)varObj.DataType;
                    string datatype = dt.Name.BaseIdentifier.Value;
                    int varSize = 0;
                    if (dt.Parameters.Count>0)
                    {
                        var pm = dt.Parameters[0];
                        if (pm is IntegerLiteral) {
                            varSize = int.Parse(pm.Value);
                        }
                        if (pm is MaxLiteral) {
                            varSize = 1024;
                        }
                    }
                    varInfo v = new varInfo
                    {
                        LineNumber = varObj.StartLine,
                        ColumnNumber = varObj.StartColumn,
                        TimesVariableUsed = 0,
                        VarType = "V",
                        DataType = datatype,
                        DataSize = varSize,
                    };
                    Variables.Add(varObj.VariableName.Value.ToString(), v);
                }
                base.ExplicitVisit(node);
            }
```

We create a dictionary and structure to hold information extracted from the **`DeclareVariableElement`**. Each time a declaration is found, it will be added to the dictionary.

## Parameter definitions

The parameters passed to the procedure will be part of the **`CreateProcedureStatement`**, so we can create a **`Visit`** reference to that as well. The code is similar to the declaration code, except we are using a different source and specifying the **`var`** type as a **`P`** for parameter. Listing 21 shows the **`Explicit`** class override for the **`CreateProcedureStatement`**.

*Code Listing 21: Create procedure explicit visit*

```
        /// <summary>
        /// Extract procedure name and any parameters
        /// </summary>
        /// <param name="node"></param>
        public override void ExplicitVisit(CreateProcedureStatement node)
        {
             ProcedureName =
node.ProcedureReference.Name.BaseIdentifier.Value;
            if (node.ProcedureReference.Name.SchemaIdentifier is not null)
            {
                SchemaName =
node.ProcedureReference.Name.SchemaIdentifier.Value;
            }
            if (node.Parameters.Count > 0)
                {
                    foreach (ProcedureParameter p in node.Parameters)
                    {
                        SqlDataTypeReference dt =
(SqlDataTypeReference)p.DataType;
                        string datatype = dt.Name.BaseIdentifier.Value;
                        int varSize = 0;
                        if (dt.Parameters.Count > 0)
                        {
                            var pm = dt.Parameters[0];

                            if (pm is IntegerLiteral)
                            {
                                varSize = int.Parse(pm.Value);
                            }
                            if (pm is MaxLiteral)
                            {
                                varSize = 1024;
                            }
                        }

                        varInfo v = new varInfo
                        {
                            LineNumber = p.StartLine,
                            ColumnNumber = p.StartColumn,
                            TimesVariableUsed = 0,
```

```
                        VarType = "P",
                        DataType = datatype,
                        DataSize = varSize,
                    };
                    Variables.Add(p.VariableName.Value.ToString(), v);
                }
            }
            base.ExplicitVisit(node);
        }
```

In this method, we are adding the parameters to our variable collection and also grabbing the procedure and schema name. If a schema is not specified, the schema name string will be empty.

## Variable usage

We can create another **`Visit`** to be called each time a variable is referenced. The code is shown in Code Listing 22.

*Code Listing 22: Visit method for declared variables*

```
    public override void ExplicitVisit(VariableReference node)
      {
          base.ExplicitVisit(node);
      }
```

The **`VariableReference`** class has a property called **`Name`**, which is the variable being referenced. Code Listing 23 shows the **`Visit`** updated to increment the **`TimeVariableUsed`** count for the variable.

*Code Listing 23: Count references to variable*

```
      public override void ExplicitVisit(VariableReference node)
      {
          if (variables.ContainsKey(node.Name))
          {
              varInfo v = variables[node.Name];
              v.TimesVariableUsed++;
              variables[node.Name] = v;

          }
          base.ExplicitVisit(node);
      }
```

When the program is finished, the **Variables** property will contain all declared variables and the number of times the variable was used. We can also add a property to the **Visitor** class to return a list of unused variables, as shown in Listing 24.

*Code Listing 24: Get list of unused variables*

```
        public List<string> UnusedVariables
        {
            get
            {
                List<string> result = new List<string>();
                result = variables.
                        Where(v =>
                  v.Value.TimesUsed<1).Select(v=>v.Key).ToList();
                return result;
            }
        }
```

## Main code

With the **Visitor** class completed, we can define our code to display the variables and any errors found. Listing 25 is the main program.

*Code Listing 25: Main variable checking program*

```
private static void Main(string[] args)
{
     Console.WriteLine("ScriptDOM Parser - Newbie procedure");
     Console.WriteLine("");

     var parser = new TSql160Parser(true, SqlEngineType.All);
     IList<ParseError>? errors = null;

     StringBuilder sb = new();
     sb.AppendLine("--  Chap 6 Newbie procedure");
     sb.AppendLine("");
     sb.AppendLine("CREATE procedure [FindUsersByLanguage](@languagecode
varchar)");
     sb.AppendLine("AS ");
     sb.AppendLine("BEGIN");
     sb.AppendLine("     set nocount on");
     sb.AppendLine("     declare @numPeopleFound int");
     sb.AppendLine("     select * from [person]");
     sb.AppendLine("     where LanguageID is null or LanguageID <>
@languagecode");
     sb.AppendLine("END");
```

```
    byte[] byteArray = Encoding.ASCII.GetBytes(sb.ToString());
    MemoryStream stream = new(byteArray);
    StreamReader rdr = new(stream);
    tSqlScript tree = (tSqlScript)parser.Parse(rdr, out errors);

    Visitor treedata = new Visitor();
    tree.Accept(treedata);

    if (errors.Count > 0)
    {
        foreach (ParseError err in errors)
        {
            Console.WriteLine(err.Message);
        }
    }
    else
    {
       // Show the parameters and variables found
       Console.WriteLine("Procedure: " + treedata.ProcedureName);
       if (string.IsNullOrEmpty(treedata.SchemaName))
          {
            Console.WriteLine("ERROR: Missing schema name");
          }
          Console.WriteLine("Parameters");
          Console.WriteLine("");
          string msg = "";
          foreach (var item in treedata.Variables.
                   Where((v) => v.Value.VarType == "P"))
          {
             msg = "";
             Visitor.varInfo v = item.Value;
             if (v.DataType.ToLower().Contains("char") && v.DataSize ==0)
             {
                 msg =  "ERROR: No size specified (defaults to 1)";
             }
             if (v.TimesUsed<1)
             {
                 msg =  "ERROR: "+item.Key+" is never used...";
             }
             Console.WriteLine(item.Key + " [" + v.DataType + "] " + msg);
          }
          Console.WriteLine("");
          Console.WriteLine("Variables");
          Console.WriteLine("");
          foreach (var item in treedata.Variables.
                 Where((v) => v.Value.VarType == "V"))
          {
              msg = "";
              Visitor.varInfo v = item.Value;
```

```
                if (v.DataType.ToLower().Contains("char") && v.DataSize ==
0)
                {
                   msg = "ERROR: No size specified (defaults to 1)";
                }
                if (v.TimesUsed < 1)
                {
                    msg = "ERROR: " + item.Key + " is never used...";
                }
                Console.WriteLine("Line " + v.LineNumber.ToString() + ": " +
                       item.Key + " [" + v.DataType + "] " + msg);
            }
            Console.WriteLine("");
        }
        Console.ReadKey();
        Console.WriteLine("");
        Console.WriteLine("End of demo");
    }
```

When this is run on the newbie's code, the console output appears as shown in Figure 9.

```
Procedure: FindUsersByLanguage
ERROR: Missing schema name when creating the procedure
Parameters

@languagecode [varchar] ERROR: No size specified (defaults to 1)

Variables

Line 7: @numPeopleFound [int] ERROR: @numPeopleFound is never used...


End of demo
```

*Figure 9: Newbie's errors*

## Summary

We hope this whets your appetite for the type of analysis ScriptDOM is capable of. You can loop through all stored procedures in a database and report the errors from this **`Visitor`** class. The source code for all chapters is available at GitHub.

# Chapter 7 Select Statement

The **SELECT** statement is certainly the most comprehensive and flexible data manipulation language statement in the SQL command set. The **SELECT** visitor class parses the **SELECT** statement, identifying tables, **where** expressions, columns, etc. The **SELECT** statement in SQL (particularly the **WHERE** expression) can be very complex, and ScriptDOM shines in helping parse it out.

We will use the **SELECT** statement to deal with the phone list problem from Chapter 1 (shown in Listing 26), but the **SELECT** statement processing is much more capable.

*Code Listing 26: PhoneList*

```
CREATE procedure [dbo].[PhoneList] (@whichparty varchar(12))
as
begin
      select * from dbo.voters
      where upper(party_affiliation) = @whichparty

end
```

## Visitor method

The visitor method call to get **SELECT** statements is shown in Listing 27. The node has a base class of a **QueryExpression**, but we are going to get the child class, **QuerySpecification**, to get the necessary extra properties for handling the **SELECT** statement. This will be a fairly common theme in your **Visitor** classes: many properties are only exposed in the child class.

*Code Listing 27: Visitor method for select statements*

```
public override void ExplicitVisit(SelectStatement node)
{
    if (node.QueryExpression is QueryExpression)
    {
       QuerySpecification qe = node.QueryExpression as QuerySpecification;
    }
    base.ExplicitVisit(node);
}
```

***Note: We need to cast the `QueryExpression` to a `QuerySpecification` to access the various properties. For example, the `QueryExpression` only has the `ForClause` property, the `OffsetClause`, and the `OrderByClause`. The `QuerySpecification` adds the `FromClause`, the `GroupByClause`, etc. It is derived from the `QueryExpression` class, but adds the additional properties needed to process a SQL `SELECT` statement.***

The node will contain all the standard properties (**FirstTokenIndex**, **FragmentLength**, **StartLine**, etc.) However, Table 8 lists the properties unique to the **SELECT** statement, which are the properties we will likely use during our analysis method.

*Table 8: Unique SELECT statement properties*

| Property | Type | Description |
|---|---|---|
| **ComputeClause** | Collection of **Compute** expressions | Although the property still exists, it has not been supported since SQL 2012. You should replace it with **ROLLUP**. |
| **Into** | Schema object name | The name of the table to which the results are sent. This can be null or possibly a fully qualified table name |
| **OptimizerHints** | Collection of optimizer hint options | Any optimizer hints found in the query (See the OPTION keyword for details)<br>• Hint kind (Enum of the option)<br>• Value (Setting for the option) |
| **QueryExpression** | Query specification | Details of tables, **WHERE** clauses, **OrderBy**, etc. |

The **QueryExpression** property has the various information we need to parse the **SELECT** statement. If a particular clause, such as **ORDER BY**, is not present in the code, it will be null in the **QueryExpression** property. This is the workhorse property of the **SELECT** statement. Table 9 lists the key properties.

*Table 9: Query expression properties*

| Property | Type | Description |
|---|---|---|
| **ForClause** | Collection of **FOR** settings (XML, JSON) | The **Options** property will vary depending on the type of FOR option. For example, the expression<br>**FOR JSON PATH ROOT('voters')**<br>will return two JSON **FOR** clause options:<br>• Path (no parameters)<br>• Root (Parameter value of voters) |
| **FromClause** | From clause | Table references:<br>• Named table reference<br>• Qualified join<br>• Null if no table specified |

| Property | Type | Description |
|---|---|---|
| **`GroupByClause`** | Group by clause | A group-by option (none, rollup, cube) collection of expression-grouping specification objects. |
| **`HavingClause`** | Having clause | The search condition (Boolean comparison class) for the having expression is found in this object |
| **`OrderByClause`** | Order by clause | This class has a list of expressions making up the order clause; the expression can be functions, column names, or integer values |
| **`SelectElements`** | List | Each column type (*, column, etc.):<br>• Select star expression<br>• Select scalar expression<br>  ○ Column reference<br>  ○ Binary expression |
| **`TopRowFilter`** | Top row filter class | Contains the:<br>• Expression (integer or function)<br>• Percent Boolean<br>• With ties Boolean |
| **`WhereClause`** | Where clause | The search condition class provides the details of the **`WHERE`** condition |

A very simple code snippet might check to see if the **`TopRowFilter`** is specified (not null) while the **`OrderByClause`** is null. Such a condition could bring back random rows, which is probably not what is expected.

## Determining tables

The **`FromClause`** property of the *query expression* contains the table references used by the query. For a simple query (one table), the **`TableReferences`** collection will contain a named table object with two key properties: the alias (null if not used) and the schema object, which provides the name of the table (property base identifier). A schema object has other properties to identify each element in a four-part table reference, shown in Table 10.

*Table 10: Four-part table reference*

| Element | Example | Property |
|---|---|---|
| Server name | **`[JDB01]`** | **`ServerIdentifier`** |

| Element | Example | Property |
|---|---|---|
| Database name | **[BookExample]** | **DatabaseIdentifier** |
| Schema name | **[dbo]** | **SchemaIdentifier** |
| Base identifier | **[customer]** | Table or view name |

To make table naming easier, we can add the private method shown in Listing 28 to our **Visitor** class. It takes a schema object and returns a string table name (with as many components as found in the object)

*Code Listing 28: Schema to TableName*

```
   private string SchemaToTableName(SchemaObjectName schemaObject)
   {
       string tblName = "";
       if (schemaObject.ServerIdentifier != null)  {
           tblName = "["+ schemaObject.ServerIdentifier.Value+"].";
           }
       if (schemaObject.DatabaseIdentifier != null) {
           tblName += "[" + schemaObject.DatabaseIdentifier.Value + "].";
           }
       if (schemaObject.SchemaIdentifier!= null) {
           tblName += "[" + schemaObject.SchemaIdentifier.Value + "].";
           }
       tblName += "[" + schemaObject.BaseIdentifier.Value + "]";
       return tblName;
 }
```

## Handling JOINS

When there are joins in the **SELECT** statement, the **Table References** collection will not be a named table object, but rather a qualified join object. This object contains properties to determine the join tables and the expression to join the tables or views.

Table 11 shows the properties of the qualified join object.

*Table 11: Qualified Join class*

| Property | Type | Description |
|---|---|---|
| **FirstTableReference** | Named table reference | Details of the table or view |
| **QualifiedJoinType** | Enumeration | Full, inner, left, right |
| **SearchCondition** | Boolean comparison | Expression used to join tables |
| **SecondTableReference** | Named table reference | Details of the second table or view |

If there are more than two tables being joined, the second table reference will be a qualified join object instead of a named table reference. Depending on the query complexity, this object structure can get quite deep.

To simplify the processing, we can create a nested object called **`JoinedTables`** and a list of joined tables in the **`Visitor`** class. Listing 29 shows the object and list property.

*Code Listing 29: Joined tables*

```
            public class JoinedTables
            {
                public string? LeftTable { get; set; }
                public string? RightTable { get; set; }
                public QualifiedJoinType? joinType { get; set; }
               public string? JoinExpression { get; set; }
                public BooleanComparisonExpression? joinOn { get; set; }
            }
            public List<JoinedTables> tables = new List<JoinedTables>();
            public string? BaseTable;
```

We can now add a function that will process the **`SELECT`** statement and return a list of joined tables found in the **`SELECT`** statement. It will also set the base table string to a qualified table name. Listing 30 is the code to iterate the **`FROM`** clause to create the joined list.

*Code Listing 30: Populate joined tables*

```
private void BuildJoinedTables(FromClause fromClause)
{
    tables.Clear();
    BaseTable = "";
    if (fromClause != null)
    {
       // If first reference is a table, rather than a join reference,
       // then only a single table
      if (fromClause.TableReferences[0] is NamedTableReference)
      {
         NamedTableReference tb =
                (NamedTableReference)fromClause.TableReferences[0];
         BaseTable = SchemaToTableName(tb.SchemaObject);
         return;
      }
      if (fromClause.TableReferences[0] is QualifiedJoin)
      {
       QualifiedJoin qn = (QualifiedJoin)fromClause.TableReferences[0];
       NamedTableReference tb1 =
(NamedTableReference)qn.FirstTableReference;
       BaseTable = SchemaToTableName(tb1.SchemaObject);
       if (tb1.Alias != null)
       {
```

```
            BaseTable += " " + tb1.Alias.Value;
        }
        NamedTableReference tb2 =
(NamedTableReference)qn.SecondTableReference;
        string SecondTable = SchemaToTableName(tb2.SchemaObject);
        if (tb2.Alias != null)
        {
            SecondTable += " " + tb2.Alias.Value;
        }

        JoinedTables jt = new JoinedTables();
            jt.LeftTable = BaseTable;
            jt.RightTable = SecondTable;
            jt.ExpObj = (BooleanComparisonExpression)qn.SearchCondition;
            jt.JoinExpression = SearchExpressionToText(jt.ExpObj);
            jt.joinType = qn.QualifiedJoinType;

            tables.Add(jt);
            return;
    }
  }
}
```

We also include a function to convert the **`Join`** expression to text to make it more readable. Listing 31 contains the **`SearchExpressionToText()`** function.

*Code Listing 31: Search Expression to Text*

```
private string SearchExpressionToText(BooleanComparisonExpression ExpObj)
{
    string ans = "";
    // Column reference, alias, and column name
    if(ExpObj.FirstExpression is ColumnReferenceExpression)
    {
        ColumnReferenceExpression? cr1 = ExpObj.FirstExpression as
ColumnReferenceExpression;
        for(int x=0;x<cr1.MultiPartIdentifier.Count;x++)
        {
            if (x>0) {  ans += "."; }
            ans += cr1.MultiPartIdentifier[x].Value;
        }
        switch(ExpObj.ComparisonType)
        {
            case BooleanComparisonType.LessThan:
                ans += " < ";
                break;
            case BooleanComparisonType.GreaterThan:
                ans += " > ";
```

```
                break;
            case BooleanComparisonType.GreaterThanOrEqualTo:
                ans += " <= ";
                break;
            case BooleanComparisonType.LessThanOrEqualTo:
                ans += " <= ";
                break;

            default:
                ans += " = ";
                break;
        }
        ColumnReferenceExpression? cr2 = ExpObj.SecondExpression as
                                    ColumnReferenceExpression;
        for (int x = 0; x < cr2.MultiPartIdentifier.Count; x++)
        {
            if (x > 0) { ans += "."; }
            ans += cr2.MultiPartIdentifier[x].Value;
        }
    }
    return ans;
}
```

## WHERE clause

The primary property on the **WHERE** clause is the **SearchCondition** object, which returns the first and second expressions. For a simple **WHERE** (one condition), the object type is a **BooleanComparisonExpression**. In the **WHERE** filter (such as **upper(party_affiliation) = @whichparty**), the **SearchCondition** object will contain a first expression (in this case, a function call object) and a second expression (a variable reference). The comparison type (equals) shows how the two expressions are compared. Table 12 shows the mapping between the code and the object.

*Table 12: Example WHERE parsing*

| SQL code | Object |
|---|---|
| **Upper(party_affiliation)** | Function call object, with a function name nested object |
| **=** | Comparison type (equals) |
| **@whichparty** | Variable reference, with a name field holding the variable name |

The object types will vary, such as column reference or string literal. The comparison type will be an enumeration, such as equals, greater than, or less than.

When there are multiple expressions, the search condition changes a bit. It will be a type Boolean binary expression, with a binary expression type of **`AND/OR`**. The first expression could be a Boolean comparison expression, with options similar to Table 12. Similarly, the second expression can be a simple comparison.

Depending on the complexity of the **`WHERE`** clause, the object can get pretty deep. Table 13 shows a visual reference of the object structure for the following SQL clause:

```
Party affiliation=@whichParty AND gender='F'
```

*Table 13: Multiple condition WHERE example*

| Object |
| --- |
| • Search condition is a Boolean binary expression<br>• Binary expression type: And<br>• First expression: Boolean comparison expression<br>  ○ First expression: Column reference (party affiliation)<br>  ○ Comparison type: Equals<br>  ○ Second expression: Variable (@whichParty)<br>• Second expression: Boolean comparison expression<br>  ○ First expression: Column reference (gender)<br>  ○ Comparison type: Equals<br>  ○ Second expression: String literal (F) |

If a third condition were added, the second expression would become a Boolean binary expression object and would have first and second expression objects. The collection of expression properties will be as deep as needed to handle all of the conditions.

***Tip: If you find the nesting getting pretty deep, it might be worth redesigning the query.***

## ORDER BY clause

The **`ORDER BY`** clause object only has one property of interest, the **`OrderByElements`** property, which is a collection of each expression in the **`ORDER BY`** clause. This list contains any number of **`ExpressionWithSortOrder`** objects.

Within the **`SortOrder`** objects, there are two interesting properties: The **`SortOrder`** is an enumerated value of ascending, descending, or not specified; the **`Expression`** property provides the details of the sort item.

### Column reference expression

A column reference uses the multi-part identifier to return the column and possibly the table or alias name. You can loop through the property to determine the column for the **`OrderBy`**. Code Listing 32 shows the code to assemble the column reference.

*Code Listing 32: Build Column reference*

```
string ans = "";
 for(int x=0;x<cr1.MultiPartIdentifier.Count;x++)
     {
        if (x>0) {  ans += "."; }
        ans += cr1.MultiPartIdentifier[x].Value;
     }
```

### Integer literal

SQL Server allows you to use numeric position when specifying sort order, although it is discouraged and may be unsupported in future SQL versions. If you use an integer sort column reference, the **`Value`** property will contain the numeric value.

***Note: You can also sort on other literal values (dates, strings, etc.). While SQL supports this, it is not likely to provide any benefit to sort on a literal string.***

## GROUP BY clause

The **`GROUP BY`** clause object contains details about the grouping logic of the query. Table 14 shows the key properties of the class.

*Table 14: Group By clause*

| Property | Type | Description |
|---|---|---|
| **`All`** | Boolean | The **`All`** option is noncompliant and only for backward compatibility |
| **`GroupByOption`** | Enumeration | Cube, None, Rollup |
| **`GroupingSpecifications`** | Collection | Collection of expression group specification objects for each value in the group by expression |

***Note: The `All` option is noncompliant for SQL Server and Azure.***

Each **ExpressionGroupSpecification** object represents a column in the **GROUP BY** clause. The key property is the expression object, which provides details about the column.

*Table 15: Expression class*

| Property | Type | Description |
|---|---|---|
| **ColumnType** | Enum | Regular or identity are the most common types |
| **Multi-Part Identifier** | Multi-part identifier | The **Identifiers** collection lists each column, and the **Value** property contains the column name |

## Having clause

The key property of the **Having** clause is a search condition (which is a Boolean comparison expression). This operates in the same manner as the search condition on the **WHERE** clause.

# Example

Figure 10 shows a **SELECT** statement and the ScriptDOM objects that contain each clause.

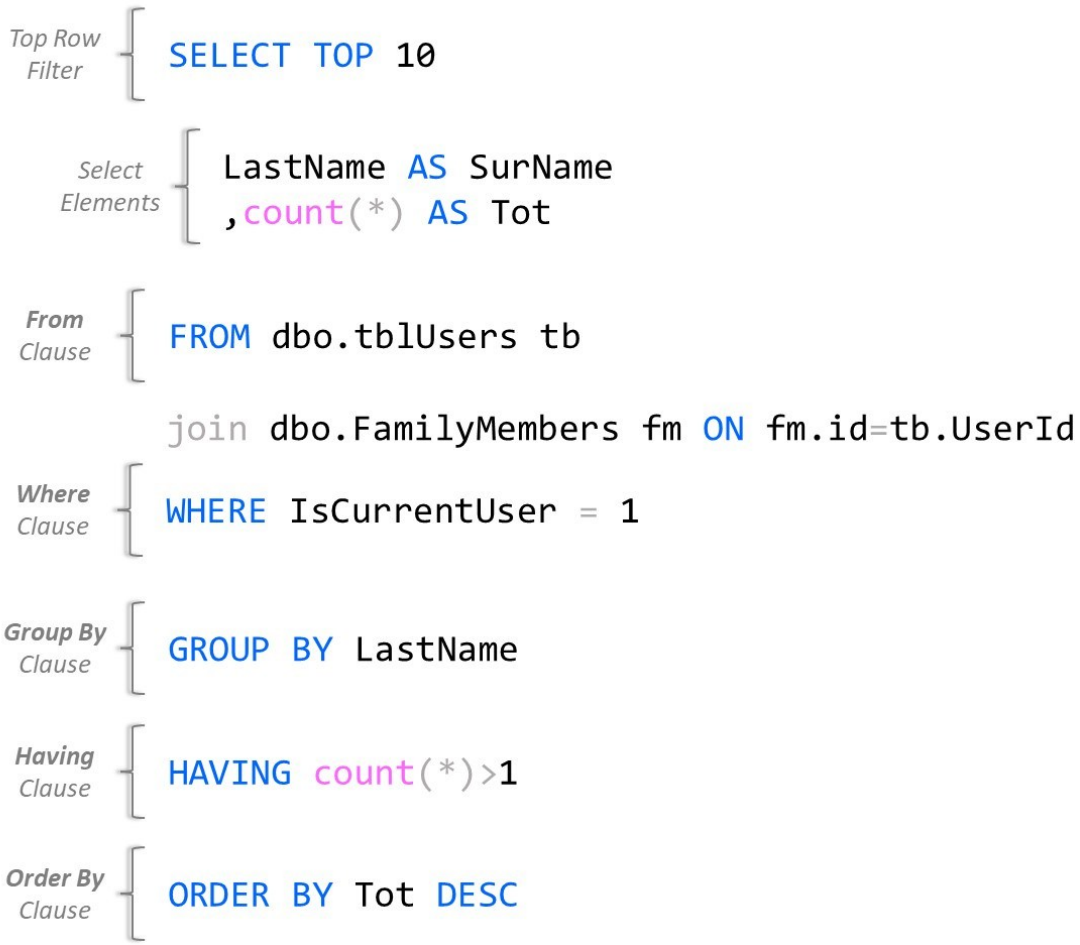


*Figure 10: SELECT statement*

## Summary

The **`SELECT`** clause can get very complex with many options. ScriptDOM allows you to extract all components of the statement. When a clause does not exist in the statement, the class will be null in the **`QuerySpecification`** object.

You can look for many issues in the select statement node, such as using **`SELECT *`**, numeric **`ORDER BY`** expressions, and tables without schemas.

# Chapter 8 Insert Statement

The **INSERT** statement in SQL is used to add rows to an existing table. The **INSERT** visitor class parses the statement, determining the table, the columns, the values or expressions, and so on.

We will work with the **INSERT** visitor class to see if we can identify the error from Glen's code to add a new company, shown in Listing 33.

*Code Listing 33: Add new company*

```
CREATE procedure [dbo].[InsertCompany] (@companyName varchar(50),@zipCode
varchar(10) )
as
begin
      DECLARE @companyID int        -- Record # of new company

      INSERT INTO dbo.Company (CompanyName,PostalCode)
      VALUES (@companyName,@zipcode)

      SET @companyID = @@IDENTITY

      UPDATE dbo.Company
      SET city = zp.City,StateCode = zp.StateCode
            FROM (select * from dbo.ZipCodes WHERE zipCode=@zipCode) zp
      WHERE CompanyID = @companyID
end
```

The visitor method call to get **INSERT** statements is shown in Listing 34.

*Code Listing 34: Visitor method for insert statements*

```
        public override void ExplicitVisit(InsertStatement node)
        {
            base.ExplicitVisit(node);
        }
```

The node will contain all the standard properties (first token index, fragment length, start line, and so on). However, Table 16 lists properties unique to the **INSERT** statement, which are the properties we will likely use during our analysis method.

*Table 16: Unique INSERT statement properties*

| Property | Type | Description |
|---|---|---|
| Insert specification | Insert specification | Class with **INSERT** details:<br>• Insert source (values, select, or exec) |

| Property | Type | Description |
|---|---|---|
| | | • Target table<br>• Top row filter |
| Optimizer hints | List | Any optimizer hints found in the query, with the **`OPTION`** keyword |

# Insert specification

This property contains all the details to evaluate the **`INSERT`** statement. The key properties are described in the sections that follow.

## Columns

This is a collection of column objects corresponding to the column list. If this collection is empty, this indicates the code does not provide a column list. In general, this is a bad practice because if a column is added or removed, or its position changed, the procedure that is dependent upon that table will now break.

Listing 35 shows how to get the **`InsertSpecification`** property and check for a missing column list.

*Code Listing 35: Visitor method for INSERT statements*

```
public override void ExplicitVisit(InsertStatement node)
{
    if (node.InsertSpecification != null)
    {
        // Check for missing column list
        InsertSpecification? obj = node.InsertSpecification;
        if (obj.Columns.Count < 1)
        {
            Console.WriteLine("INSERT statements should always " +
                              "include a column list");
        }
    }
    base.ExplicitVisit(node);
}
```

If there are columns in the collection, you can iterate the collection to get details about the columns to be inserted. Each item in the collection is a column reference expression. The key properties for the **`Column`** are described next.

### Collation

The **`Collation`** property is an identifier object, and the **`Value`** property provides the name of the collation for the column. If no collation exists in the column, this value will be null.

### Column type

The column type is an enumeration value used for various statements. The most likely column types on the **`insert`** statement are shown in Table 17.

*Table 17: Column type enumeration*

| Enum Name | Description |
|---|---|
| **`Regular`** | A standard column |
| **`Identity`** | An identity key column |

### Multi-part identifier

Each column has a multi-part identifier property, which can be used to determine the name of the column being inserted. The property contains a **`count`** field and a collection of **`Identifier`** objects. The **`Value`** property in each **`Identifier`** has the name of the database field.

## Insert source

The **`InsertSource`** can be a different class, depending on the data being inserted. The three possible class types are shown in Table 18.

*Table 18: Insert Source classes*

| Class | Description |
|---|---|
| **`ValuesInsertSource`** | Standard values clause, list of values to add |
| **`SelectInsertSource`** | The data is being inserted from a **`SELECT`** clause |
| **`ExecuteInsertSource`** | A stored procedure that is being executed to provide data |

### ValuesInsertSource

This class has two key properties. The **`IsDefaultValues`** Boolean indicates if all the columns are default values. The primary property is the **`RowValues`** property, a collection of **`RowValue`** objects. This allows you to parse all rows if a multirow insert is in the code.

Within each **`RowValue`** element is a **`ColumnValues`** collection. Some of the most common class types are shown in Table 19.

*Table 19: Column value types*

| Class | Description |
|---|---|
| Variable reference | A variable, such as **`@companyName`**<br>The **`Name`** property provides the actual variable name |
| Function call | A SQL standard or UDF function, such as **`GetDate()`**<br>The function's **`Name`** property has details about the function, such as **`Parameters`** and the **`Value`** property, with the function name |
| String literal | A string of characters<br>This class has properties **`Is National`** (i.e. nvarchar text string), **`Is Large Object`**, and the **`Value`** property for the actual text |
| Integer literal | A numeric integer value<br>The value property contains the text (even though numeric), but the property is still a string |

## SelectInsertSource

When the object is a **`SelectInsertSource`**, the key property will be a query specification (see Chapter 7 for details on the query specification class). You can use the query parsing code to grab the name of the table, the top expression, the **`WHERE`** clause, and so on.

## ExecuteInsertSource

When the object is an **`ExecuteInsertSource`**, the key property will be the **`Execute`** property, which contains an **`ExecuteSpecification`** object. The key properties for the **`Execute`** class are shown in Table 20.

*Table 20: ExecuteSpecification properties*

| Property | Type | Description |
|---|---|---|
| **`ExecutableEntity`** | Execute procedure reference | Includes the procedure reference object and any parameters used |
| **`ExecuteContent`** | Execute content object | The **`Principal`** property will contain the name of the user or login to execute the procedure as. The **`Kind`** can be **`User`** or **`Login`**. |
| **`LinkedServer`** | Identifier | The **`Identifier`** object provides the name of the linked server the procedure is found on |

## Target

The **Target** property is a named table reference object, which you can use to determine the table to insert the values into. Table 21 lists the properties of the schema object in the table reference.

*Table 21: Schema object reference*

| Element | Example | Property |
|---|---|---|
| Server name | **[JDB01]** | Server identifier |
| Database name | **[BookExample]** | Database identifier |
| Schema name | **[dbo]** | Schema identifier |
| Base identifier | **[customer]** | Table or view name |

You can use the **Target** property to determine the table the **insert** statement applies to.

# SetVariableStatement

In addition to the **INSERT** statement, we are also going to add a visitor method for the **SetVariable** command. For this example, we are specifically looking for references to **@@identity** and suggesting **SCOPE_IDENTITY** instead. Listing 36 shows the visitor method.

*Code Listing 36: Set Variable Statement*

```
public override void ExplicitVisit(SetVariableStatement node)
{
     base.ExplicitVisit(node);
}
```

The **SetVariableStatement** class has several properties we can use to determine the variable being assigned and the content to be set in the variable

*Table 22: Unique SetVariable statement properties*

| Property | Types | Description |
|---|---|---|
| **Expression** | Global variable expression | The expression's name property will indicate the name of the global variable |
| | Function call | The function name property is an identifier object, and the identifier object has a property called **Value**, with the name of the function being called |

| Property | Types | Description |
|---|---|---|
| | Integer, string literal | If a literal value (string, integer, and so on), the **`Value`** property will contain the value being set |
| **`Variable`** | Variable reference | Provides the name of the variable being updated |

In our code, we are going to check to see if **`@@identity`** was used, and if we detect it, suggest using **`SCOPE_IDENTITY()`** instead. Listing 37 shows the revised code.

*Code Listing 37: Check for @@identity*

```
  // Check to see if @@identify is used
  public override void ExplicitVisit(SetVariableStatement node)
  {
      if (node.Expression is GlobalVariableExpression)
      {
         GlobalVariableExpression exp =
(GlobalVariableExpression)node.Expression;
         if (exp.Name.ToLower()=="@@identity")
            {
                Console.WriteLine("Replace @@identify with
SCOPE_IDENTITY()");
            }
      }
      base.ExplicitVisit(node);
 }
```

When the code is now analyzed, it will identify the issue with **`@@identity`** and suggest the replacement. This will fix Glen's code that broke once another developer added a trigger to a related table.

# Example

An example **`INSERT`** statement is shown in Figure 11.


*Figure 11: Example INSERT statement*

## Summary

The **INSERT** statement has several properties and sources of the content to insert. You can use this visitor method to detect a missing column list, use of **@@identity**, and so on. There are other issues you can detect, such as updating an identity column without an identity insert.

# Chapter 9 Update Statement

The **UPDATE** statement is used to update columns in a database table. We can create a visitor method to grab the **UPDATE** statement. In this chapter, we are going to write code to check if an **UPDATE** statement contains a **WHERE** clause. However, if the **UPDATE** statement is a temporary table or table variable, we won't flag it as an error. The following code snippet can create a visitor method to grab **UPDATE** statements.

```
public override void Visit(UpdateStatement node)
```

The node parameter will be an **UpdateStatement** type, and the key property will be the update specification object. If a clause is not found in the statement, the property for that clause will be null.

Listing 38 shows a sample stored procedure that updates tables.

*Code Listing 38: Update statements*

```
CREATE PROCEDURE dbo.UpdateCustomers(@taxRate float,@statecode varchar(2))
AS
BEGIN
    UPDATE dbo.Customer
           SET taxRate = @taxRate,
               LastUpdated = getDate(),
               UpdCounter += 1
    WHERE stateCode=@stateCode

END
```

## Target

The **Target** property is a **NamedTableReference** object, which can be used to determine the table being updated. The code snippet in Listing 39 shows how we can test for a missing **WHERE** clause, and that the table is not a temporary table.

*Code Listing 39: Check target name if missing WHERE clause*

```
// Search for update statements with no WHERE clause
UpdateSpecification? upd = node.UpdateSpecification;
if (upd.WhereClause is null && upd.Target is NamedTableReference)
  {
     NamedTableReference tbl = (NamedTableReference)upd.Target;
     string tblName = tbl.SchemaObject.BaseIdentifier.Value;
```

```
    if ( ! ((tblName.StartsWith("#") || tblName.StartsWith("@")) ) )
         {
             Console.WriteLine("Line "+node.StartLine.ToString()+
                    ":  UPDATE to " + tblName +
                    " table is missing a WHERE expression");
         }

 }
```

The **Schema.Object.BaseIdentifier** provides the table name being updated. In addition, the server identifier, database identifier, and schema identifier can access each part of the table reference. Figure 12 illustrates a fully referenced table and properties to access each part.

**[s11].[jbsite].[dbo].[Person]**

*Figure 12: Fully qualified table*

## FROM clause

The **UPDATE** statement allows you to update the data from another table, as shown in Listing 40.

*Code Listing 40: Sample UPDATE FROM*

```
UPDATE dbo.taxLog
   SET rateCode = xx.Rate
   FROM (SELECT StateCode,Rate FROM  stateCodes ) xx
   WHERE xx.StateCode=taxlog.State
```

If a **FROM** clause exists in the **UPDATE** statement, the **FROM** property will contain the table reference object. The **FromClause** object has a collection of **TableReference** objects, as shown in Table 23.

*Table 23: From Clause properties*

| Property | Type | Description | Properties |
|---|---|---|---|
| **Table References** | Query-derived table | Alias query expression | Alias assigned<br>• From clause<br>• Group By clause<br>• Order By clause<br>• Select elements<br>• Where clause |

| Property | Type | Description | Properties |
|---|---|---|---|
| | Named table reference | Alias schema object | Alias assigned<br>• Server name<br>• Database name<br>• Schema name<br>• Base identifier |

## Set clauses

This collection is a list of **`AssignmentSetClause`** objects, an object for each column being updated. Due to a recent requested software update, the company would like all date columns to contain UTC (Coordinated Universal Time) dates, rather than local dates. SQL Server has a function called **`GetUTCDate()`**, which will return the UTC time.

We need code to review all **`UPDATE`** statements and report those statements that update a date field and use **`GetDate()`**, rather than **`GetUTCDate()`**. We do not want to report any updates to temporary tables, though.

Each item in the collection has the properties shown in Table 24.

*Table 24: Set clause properties*

| Property | Types | Description |
|---|---|---|
| **`AssignmentKind`** | Equals (=)<br>Add equals (+)<br>Multiply equals (*)<br>Subtract equals (-) | Column = value<br>Column **+=** value<br>Column ***=** value<br>Column **-=** value |
| **`Column`** | Column reference expression | Column type: regular, identity<br>Multi-part identifier value |
| **`NewValue`** | Variable reference | Name holds variable name |
| | Function call | Function name |
| | String literal | Value property |
| | Integer literal | Value property |

Listing 41 shows code to check for the **`GetDate()`** function call, so it can be replaced with **`GetUtcDate()`**.

*Code Listing 41: Identify GetDate() updates*

```
UpdateSpecification upd = node.UpdateSpecification;
foreach ( AssignmentSetClause setvar in upd.SetClauses)
{
   if (setvar.NewValue is FunctionCall)
   {
      FunctionCall fn = setvar.NewValue as FunctionCall;
      if ( fn != null )
      {
          NamedTableReference tbl = (NamedTableReference)upd.Target;
          string tblName = tbl.SchemaObject.BaseIdentifier.Value;
          if (! ((tblName.StartsWith("#") || tblName.StartsWith("@"))) )
          {
            if (fn.FunctionName.Value.ToLower() == "getdate")
            {
                 Console.WriteLine("Line " + node.StartLine.ToString() +
                        ":  UPDATE to " + tblName +
                        " table is using GetDate() rather than
GetUtcDate())");
            }
          }
      }
    }

}
```

## WHERE clause

The primary property on the **WHERE** clause is the search condition object, which returns the first and second expressions. For a simple **WHERE** (one condition), the object type is a Boolean comparison expression. In the **WHERE** filter (such as **statecode = @statecode**), the search condition object will contain a first expression (in this case, a **FunctionCall** object) and a second expression (a variable reference). The comparison type (equals) shows how the two expressions are compared. Table 25 shows the mapping between the code and the object.

*Table 25: Example WHERE parsing*

| SQL code | Object |
|---|---|
| **Upper(party_affiliation)** | Function call object, with a function-name nested object |
| **=** | Comparison type (equals) |
| **@whichparty** | Variable reference, with a name field holding the variable name |

The object types will vary, such as column reference, string literal, and so on. The comparison type will be an enumeration, such as equals, greater than, or less than.

## Example

An example **UPDATE** statement is shown in Figure 13.

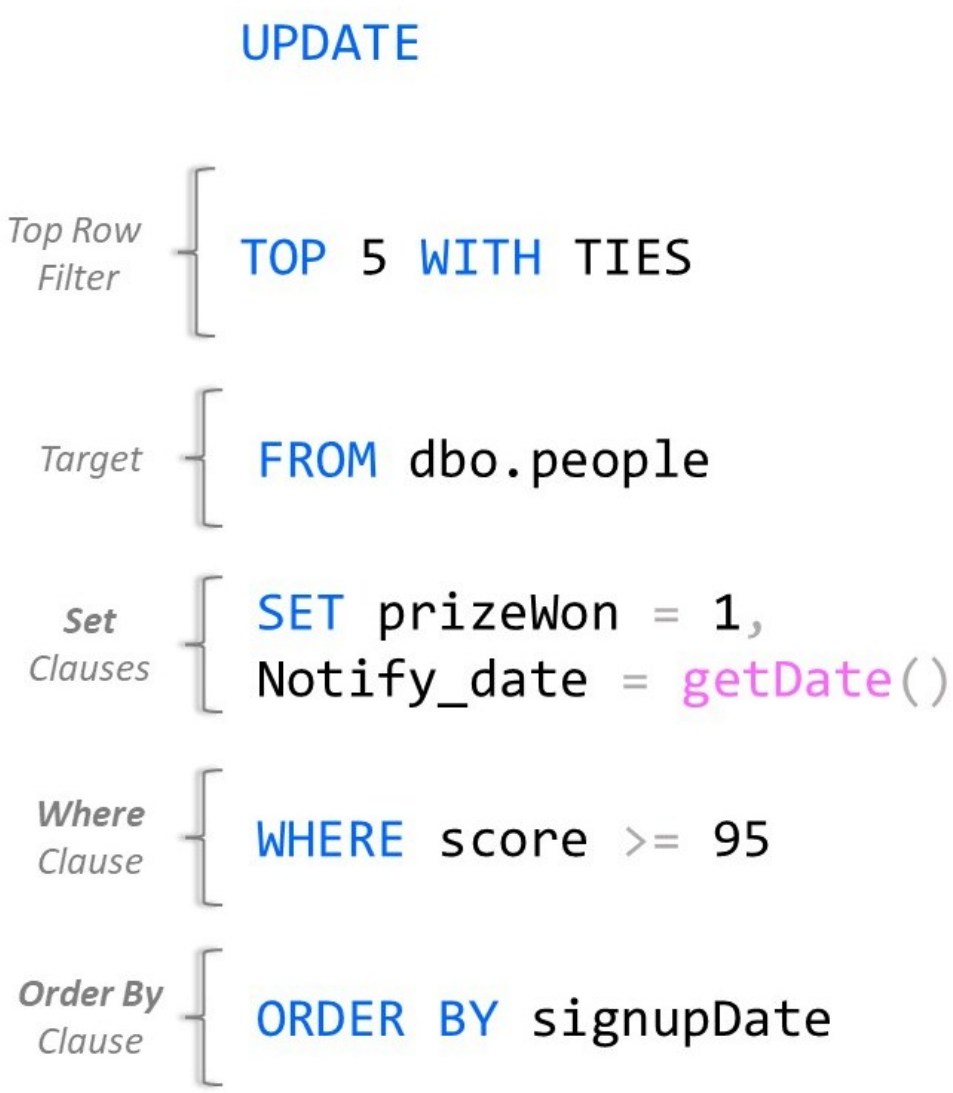


*Figure 13: Example UPDATE statement*

## Summary

The **UPDATE** statement has several properties and sources of the content to update. You can use this visitor method to search the columns being updated, determine which table is being updated, and possibly the table providing the update source.

# Chapter 10 Delete Statement

The **DELETE** statement is used to delete rows in a database table. We can create a visitor method to grab the **DeleteStatement**. In this chapter, we are going to write code to check if a **DELETE** statement contains a **WHERE** clause. However, if the **DELETE** statement is a temporary table or table variable, we won't flag it as an error.

Another consideration to check for is the combination of **TOP x** with an **ORDER BY** (which could basically randomly determine which rows to delete). The following code snippet can create a visitor method to grab **DELETE** statements.

```
public override void Visit(DeleteStatement node)
```

The node parameter will be a **DeleteStatement** type, and the key property will be the delete specification object. If a clause is not found in the statement, the property for that clause will be null.

Listing 42 shows a sample stored procedure that deletes data.

*Code Listing 42: Delete statements*

```
CREATE PROCEDURE dbo.RemoveInactiveVoters()
AS
BEGIN
      DELETE
      FROM dbo.Voters
      WHERE affiliation = '?'

END
```

## Target table

*Code Listing 43: Check target name if missing WHERE clause*

```
public override void ExplicitVisit(DeleteStatement node)
{
    DeleteSpecification? upd = node.DeleteSpecification;
    if (upd.WhereClause is null && upd.Target is NamedTableReference)
    {
        NamedTableReference tbl = (NamedTableReference)upd.Target;
        string tblName = tbl.SchemaObject.BaseIdentifier.Value;
        if (!((tblName.StartsWith("#") || tblName.StartsWith("@"))))
        {
            Console.WriteLine("Line " + node.StartLine.ToString() +
```

```
                    ":  DELETE for " + tblName +
                    " table is missing a WHERE expression");
            }
        }
        base.ExplicitVisit(node);
    }
```

## WHERE clause

The primary property of the **WHERE** clause is the search condition object, which returns the first and second expressions. For a simple **WHERE** (one condition), the object type is a Boolean comparison expression. In the **WHERE** filter (such as **affiliation = '?'**), the search condition object will contain a first expression (in this case, a column reference expression object) and a second expression (a string literal). The comparison type (equals) shows how the two expressions are compared. Table 26 shows the mapping between the code and the object.

*Table 26: Example WHERE parsing*

| SQL code | Object |
|---|---|
| **Affiliation** | Column reference expression |
| **=** | Comparison type (equals) |
| **?** | String literal |

The object types will vary, such as column reference or string literal. The comparison type will be an enumeration, such as equals, greater than, or less than.

# Example

An example **DELETE** statement is shown in Figure 14.

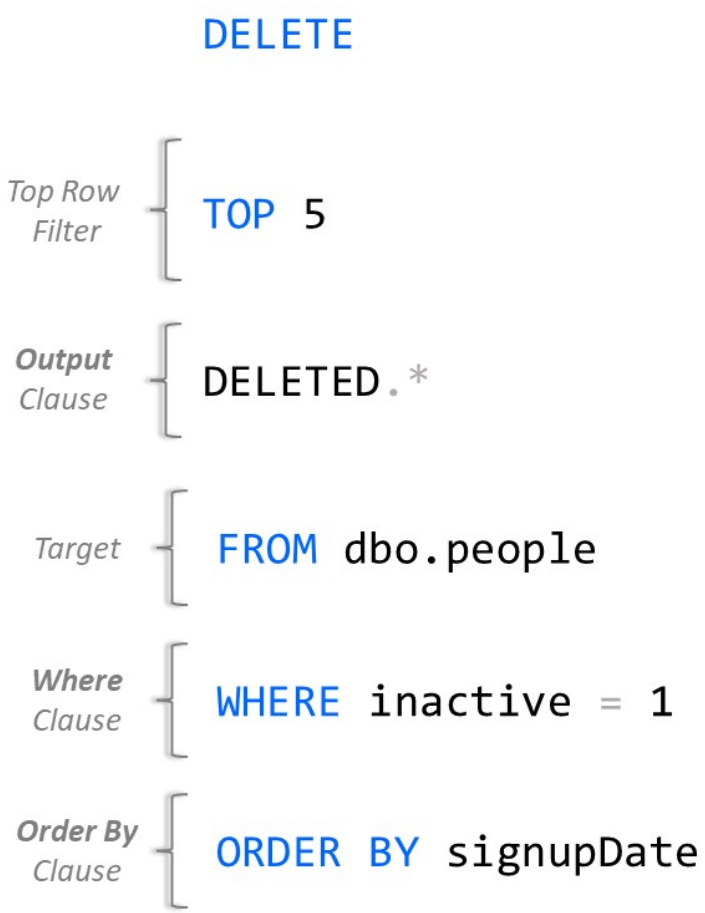


*Figure 14: Example DELETE statement*

## Summary

The **DELETE** statement has a few properties with the content of the **DELETE** command. You can use this visitor method to find the table having rows deleted, and hopefully, a **WHERE** expression, so the code doesn't delete the entire table.

# Chapter 11 Functions

SQL has a large number of functions, such as **`UPPER()`**, **`TRIM()`**, and **`CHARINDEX()`**. We can use ScriptDOM to find the functions being used in the code. In this chapter, we will look at functions found within a stored procedure.

We recently had a custom-written **`string_split()`** function (since we are using a version of SQL prior to 2016). However, due to recursion, this version was limited to 100 rows. We need to find all references to the custom-written version so we can update it to the SQL 2016 built-in version.

We can set up a **`Visitor`** class to catch all function calls, as shown in Listing 44.

*Code Listing 44: Function Call visitor*

```
public class Visitor : TSqlFragmentVisitor
{
    public override void ExplicitVisit(FunctionCall node)
    {

        base.ExplicitVisit(node);
    }

}
```

The key properties of the **`FunctionCall`** class are shown in Table 27.

*Table 27: Function Call properties*

| Property | Type | Description |
|---|---|---|
| **`FunctionName`** | Identifier | The **`Value`** property contains the function name |
| **`Parameters`** | Collection of parameters passed to the function | • Function call (nested function call)<br>• Column reference expression<br>• String literal<br>• Numeric literal |
| **`CallTarget`** | Multi-part identifier call target | If the function is a user-defined function, this property will not be null. You can iterate the identifiers to build the schema containing the function. |

## Identifying types of functions

SQL has a number of built-in functions, so we can write a visitor method that will flag each function call as SQL or a UDF. In our case, we want to make sure we use the native SQL version of **`String_Split()`**, not the UDF version.

Listing 45 shows the code to report the type of function, either SQL or UDF.

*Code Listing 45: Categorize function calls*

```
public override void ExplicitVisit(FunctionCall node)
{
    string fnName = "";
    string schema = "";
    Identifier? fn = node.FunctionName as Identifier;
    if (fn != null)  {
            fnName = fn.Value;
    }
    // This will be set for user-defined functions
    CallTarget? ct = node.CallTarget as CallTarget;
    if (ct != null && ct is MultiPartIdentifierCallTarget)
    {
            MultiPartIdentifierCallTarget ctm =
(MultiPartIdentifierCallTarget)ct;
            string ans = "";
            for (int x = 0; x < ctm.MultiPartIdentifier.Identifiers.Count;
x++)
            {
                if (x > 0) { ans += "."; }
                ans += ctm.MultiPartIdentifier.Identifiers[x].Value;
            }
            schema = ans;

    }
        if (schema == "")   {
            Console.WriteLine("SQL " + fnName);
        }
        Else                {
            Console.WriteLine("UDF " + schema+"."+fnName);
        }

        base.ExplicitVisit(node);
}
```

## Reporting function errors

As an example of a function error we can look for, we can use the **`Format()`** function. A common mistake I've seen is not understanding the case-sensitivity of the function. For example, if I want to display the current date formatted, I would use MM/dd/yyyy. Table 28 shows a couple of example date formats and what they return.

*Table 28: Date formats*

| Format | Returns | Notes |
|---|---|---|
| MM/dd/yyyy | 08/25/2023 | This is good! |
| mm/dd/yyyy | 12/25/2023 | Lower case **mm** is minutes, not month |
| MM/DD/YYYY | 08/DD/YYYY | DD and YYYY are invalid |
| MM/dd/YYYY | 08/25/YYYY | YYYY is invalid |

We can use the visitor pattern to report potential date format errors. Listing 46 shows code to potentially catch incorrect format values.

*Code Listing 46: Check date format values*

```
if (fnName.ToUpper()=="FORMAT" && node.Parameters.Count>1)
    {
        StringLiteral? fmt = node.Parameters[1] as StringLiteral;
        if (fmt != null)
        {
            string fmtString = fmt.Value;
            if (fmtString.StartsWith("mm/"))
            {
                Console.WriteLine("Check date format ("+fmtString+"),
                                  mm is for minutes, MM is for months");

            }
            if (fmtString.EndsWith("/YYYY") || fmtString.EndsWith("/YY"))
            {
                Console.WriteLine("Check date format ("+fmtString+"),
                                YYYY or YY for year must be lower case");
            }
        }
   }
```

## Some other ideas

You can use this visitor to look for other types of errors, particularly subtle ones that might slip through. Table 29 offers a few more suggested patterns to look for.

*Table 29: Possible function problems*

| Function name | Potential issue |
|---|---|
| **`CharIndex()`** | If the first string parameters contain a wildcard character (such as **`%`** or **`*`**), suggest using **`PatIndex()`** instead |
| Nested **`Replace()`** functions | Replace code like<br>**`replace(replace(Replace('[1&2]', '[','('),']',')'),'&','+')`**<br>with **`TRANSLATE('[1&2]','[&]','(+)')`** |
| **`HashBytes()`** | Replace the deprecated hash algorithm (**`MD2`**, **`MD4`**, **`MD5`**, **`SHA`**, **`SHA1`**) with the more secure **`SHA2_256`** or **`SHA2_512`** algorithm |
| **`@@Identity`** | Replace **`@@identity`** with **`Scope_Identity()`** to prevent triggers from impacting the expected identity key |

Hopefully, this list will provide you with some ideas of functions you might want to review in your code. Each of these examples shows valid function calls, but unexpected behavior could surprise a SQL developer. As another example, you might want to set a standard way to present dates when using the **`Format()`** function.

# Summary

The function call visitor allows you to easily find user-defined functions in your code, as well as suggest potential problems with the functions. This came in very handy for my company to find the **`String_Split()`** user-defined functions and replace them with the native SQL version.

# Chapter 12 Set Commands

SQL Server has a number of different **SET** options, allowing you to change the behavior of a stored procedure. In this chapter, we will look at how ScriptDOM can view those settings and their values. We will look for two conditions: the first is to find any deprecated options, and the second is to see if the **SET NOCOUNT** command is used.

## Checking for set commands

To check for the commands to set options, you need to create a visitor class that can be used as a parameter to the **Accept()** method. Listing 47 shows a shell for the **Visitor** class.

*Code Listing 47: Visitor class*

```
public class Visitor : TSqlFragmentVisitor
{
    public Visitor() {
    }
    // Visitor statements
    public override void ExplicitVisit(SetCommandStatement node)
    {
        base.ExplicitVisit(node);
    }
    public override void ExplicitVisit(PredicateSetStatement node)
    {
        base.ExplicitVisit(node);
    }
}
```

Notice that there are two statements (hence visitor methods) to review the various **SET** commands available.

## SetCommandStatement

The **SetCommandStatement** class has a key property, **commands**, which is a collection of general set command objects. Table 30 shows the properties of the **GeneralSetCommand** class.

*Table 30: GeneralSetCommand class*

| Property | Types | Description |
|---|---|---|
| **CommandType** | General set command type | This is an enumerated value of various **SET** commands. You can view various values |

| Property | Types | Description |
|---|---|---|
| | | here. This is a small set of commands where the parameter is not simply on or off.<br>**DateFormat** is an example of such a **SET** command. |
| **Parameter** | Scalar expression, such as:<br>• Identifier literal<br>• Function call | Identifier object, with the key properties:<br>• Literal type<br>• Quote type<br>• Value<br>Function call object<br>• Function name<br>• Parameters |

## PredicateSetStatement

The **PredicateSetStatement** class has two key properties, shown in Table 31.

*Table 31: PredicateSetStatement properties*

| Property | Types | Description |
|---|---|---|
| **IsOn** | Boolean | True or false, is the **SET** turned on |
| **Options** | Set options | An enumerated type indicating the **SET** option. You can see the complete list here. |

We can use this visitor method to check if **SET NOCOUNT** was found. We can use a Boolean flag (**FoundNoCount**) as a property of the visitor class. The code in Listing 48 shows how to update the flag if **SET NOCOUNT** was found and turned on.

*Code Listing 48: Check for NOCOUNT*

```
public override void ExplicitVisit(PredicateSetStatement node)
  {
      // Set NOCOUNT Check
      if (node.Options==SetOptions.NoCount)
      {
          if (node.IsOn) {  FoundNoCount= true; }
      }
      base.ExplicitVisit(node);
  }
```

We will also need a public method in the visitor to be called after the code is processed. Listing 49 shows some code to display a console message if **`SET NOCOUNT`** is missing or set to **`OFF`**.

*Code Listing 49: Final check method*

```
        public void FinalChecks()
        {
            if (!FoundNoCount)
            {
                Console.WriteLine("|The stored procedure does
                          not include a SET NOCOUNT ON statement.")
            }
        }
```

***Note: For an exercise, you can tweak Code Listing 49 to distinguish between `SET NOCOUNT` not being found and it being found but turned off.***

Other items to check for might include:

- **`ShowPlanAll`**, **`ShowPlanText`**: Should not be included in a production stored procedure
- **`Set Ansi_Nulls`**, **`Set Ansi_Padding`**: Both are deprecated **`SET`** options

# Other SET statements

Some other statements have parameters that are not represented by on/off, literal values, or function calls. Some of these have their own statements.

## SET IdentityInsert statement

The **`SET IdentityInsert`** statement has an **`IsOn`** (Boolean) property, but also a schema object name property called **`Table`**. This property is the table object that the command applies to.

## SET RowCount statement

The **`SET RowCount`** statement restricts the number of rows that a **`SELECT`** statement returns. It does not impact **`DELETE`**, **`UPDATE`**, or **`INSERT`** statements. The property **`NumberRows`** is of type **`ValueExpression`**. This can be a literal value, a global variable, or a local variable. Generally, **`Top X`** provides a better option, but **`RowCount`** can be used. If **`RowCount`** and **`Top X`** are both used, the lower value is respected.

## SET ErrorLevel statement

**`SET ErrorLevel`** has a property **`Level`**, which is a scalar expression (function call, literal value, etc.).

## Summary

The **SET** commands can change the way a procedure behaves, so you can use ScriptDOM to review the command and suggest any improvements or fixes. My company requires that all procedures use **SET NOCOUNT** in production to reduce network traffic. Using ScriptDOM, it is easy to identify those procedures that are missing the command (or set the property to **OFF**).

# Chapter 13 Dangerous Commands

There are some commands that probably should never be included in a stored procedure. In this chapter, we will look for these commands and report them as found.

## Dangerous commands

The statements **`Kill`** and **`Shutdown`** should not be hidden inside stored procedures. (Although in a properly secured system, it is unlikely the stored procedure would have the necessary rights to use these commands.) The **`WaitFor`** statement could potentially be problematic, since it could pause the procedure, and possibly keep some locks on a table and other SQL objects.

To check for these commands and others you might consider, we can create a visitor to capture code fragments. This visitor method will get called for every statement within the procedure. Listing 50 shows the visitor method to get SQL statement fragments.

*Code Listing 50: SQL fragments*

```
        public override void Visit(TSqlFragment fragment)
        {
            base.Visit(fragment);
        }
```

Since this visitor method will be called from every line in the procedure, use it sparingly. While we are introducing it here to illustrate generic use, each of the dangerous statements could also be caught using the standard visit statement calls. Table 32 lists the three statement nodes we could consider dangerous.

*Table 32: Dangerous code*

| Statement class | Description |
|---|---|
| **`Kill`** statement | This class has a parameter property containing the SPID to be killed, and a Boolean value for status-only setting. |
| **`Shutdown`** statement | Although it is very unlikely that a stored procedure would have rights to run this statement, it will literally shut down the server. The only property is a Boolean value **`WithNoWait`** on the command. |
| **`WaitFor`** statement | The **`WaitFor`** class has a property to indicate the **`WaitFor`** option (delay or time), and a parameter value indicating the time. |

***Note: These statements are hardly ever used in a stored proc, but if hackers want to wreak havoc on your database, they could hack a stored procedure and add one of these statements.***

We can modify our **`Visit`** method to check for these commands, and if so, write a console line indicating they were found and should be reviewed. Code Listing 51 shows the code to check for the commands.

*Code Listing 51: Check for certain commands*

```
        public override void Visit(TSqlFragment fragment)
        {
            if (fragment is KillStatement)
            {
                KillStatement kill = (KillStatement) fragment;
                Console.WriteLine("ERROR: Kill statement should
                      not be used within a stored procedure");
            }
            if (fragment is ShutdownStatement)
            {
                ShutdownStatement sd = (ShutdownStatement) fragment;
                Console.WriteLine("ERROR: The Shutdown statement should
                       not be used within a stored procedure");
            }
            if (fragment is WaitForStatement)
            {
                WaitForStatement wf = (WaitForStatement) fragment;
                Console.WriteLine("WARNING: The WaitFor statement can
                       impact performance, particularly if tables are
                       locked while waiting");
            }
            base.Visit(fragment);
        }
```

## Takeaway

One of the key takeaways from this chapter is the ability to create a visitor method to get every SQL statement. While doing so creates a large-case statement that defeats the purpose of the visitor pattern, it can be a handy technique when you want to find a particular visitor statement that handles a specific SQL statement.

## Summary

In a properly secured SQL system, it is very unlikely that a stored procedure could be run containing the **`Kill`** or **`Shutdown`** command. However, the **`WaitFor`** statement could be used, potentially slowing down the entire server.

This chapter shows another visitor method to get all SQL statements with a single method. Using this approach should only be used for very small code snippets. Creating a large code block to generally handle any statement defeats the benefits that the visitor methods provide.

# Chapter 14 Helper Functions

As you work with ScriptDOM, you will likely find some common functionality to use across various visitor methods. In this chapter, we will create a static library of some functions that can be useful when parsing the code.

## Helper class

We will create a static class called **`ScriptDOMHelper`** and add methods to make it easier to work with the library. Code Listing 52 shows the helper shell code.

*Code Listing 52: Helper shell*

```
using Microsoft.SqlServer.TransactSql.ScriptDom;

namespace Chap014
{
    static public class ScriptDOMHelper
    {

    }
}
```

We will add our methods to this static public class.

## IsTempTable()

This method takes a table name and returns a Boolean value if the table is a temporary table or a table variable. For statements like **`UPDATE`** and **`DELETE`**, it is risky to use these statements without a **`WHERE`** expression. However, these statements are likely acceptable when dealing with a table variable or temporary table. Code Listing 53 shows the function. You can pass it a table string of a named table reference object.

*Code Listing 53: IsTempTable()*

```
        static public bool IsTempTable(String tblName)
        {
             return tblName.StartsWith("#") || tblName.StartsWith("@");
        }
        static public bool IsTempTable(NamedTableReference tbl)
        {
            if (tbl != null)
            {
                string tblName = tbl.SchemaObject.BaseIdentifier.Value;
```

```
                return = IsTempTable(tblName);
            }
            return false;
        }
```

## ColumnType()

This method takes a column definition object and returns a string representation of the column.

*Code Listing 54: ColumnType()*

```
static public string ColumnType(ColumnDefinition col)
{
    string ans = col.ColumnIdentifier.Value;    // Get column name
    ans += " " + col.DataType.Name;             // Add the data type
    if (col.IdentityOptions != null)
    {
        ans += " Identity(" + col.IdentityOptions.IdentitySeed.ToString() +
"," +

col.IdentityOptions.IdentityIncrement.ToString() + ")";
    }
    if (col.ComputedColumnExpression != null &&
        col.ComputedColumnExpression is ScalarExpressionSnippet)
    {
        ans += " Expression=" +  ((ScalarExpressionSnippet)
                                  col.ComputedColumnExpression).Script;
    }
    return ans;
}
```

## ParserForVersion()

This method takes a string parameter containing a SQL version (2022, 2019, 2017) and returns the appropriate parser to use. Code Listing 55 shows the code.

*Code Listing 55: ParserForVersion()*

```
        static public TSqlParser ParserForVersion(string SQLVersion)
        {
            if (SQLVersion.Contains("2019")) { return new
TSql150Parser(true); }
            if (SQLVersion.Contains("2017")) { return new
TSql140Parser(true); }
            if (SQLVersion.Contains("2016")) { return new
TSql130Parser(true); }
```

```
            if (SQLVersion.Contains("2014")) { return new
TSql120Parser(true); }
            if (SQLVersion.Contains("2012")) { return new
TSql110Parser(true); }
            if (SQLVersion.Contains("2008")) { return new
TSql100Parser(true); }
            if (SQLVersion.Contains("2005")) { return new
TSql90Parser(true); }
            if (SQLVersion.Contains("2000")) { return new
TSql80Parser(true); }

            // Assume latest parser
            return new TSql160Parser(true, SqlEngineType.All);
        }
```

## SearchExpressionToText ()

This method takes a Boolean comparison expression and returns a text representation of the expression. Code Listing 56 shows the code.

*Code Listing 56: SearchExpressionToText()*

```
private string SearchExpressionToText(BooleanComparisonExpression ExpObj)
{
    string ans = "";
    // Column reference, alias, and column name
    if(ExpObj.FirstExpression is ColumnReferenceExpression)
    {
        ColumnReferenceExpression? cr1 = ExpObj.FirstExpression as
                                         ColumnReferenceExpression;
        for(int x=0;x<cr1.MultiPartIdentifier.Count;x++)
        {
            if (x>0) {  ans += "."; }
            ans += cr1.MultiPartIdentifier[x].Value;
        }
        switch(ExpObj.ComparisonType)
        {
            case BooleanComparisonType.LessThan:
                ans += " < ";
                break;
            case BooleanComparisonType.GreaterThan:
                ans += " > ";
                break;
            case BooleanComparisonType.GreaterThanOrEqualTo:
                ans += " <= ";
                break;
            case BooleanComparisonType.LessThanOrEqualTo:
```

```
                ans += " <= ";
                break;

            default:
                ans += " = ";
                break;
        }
        ColumnReferenceExpression? cr2 = ExpObj.SecondExpression as
                                    ColumnReferenceExpression;
        for (int x = 0; x < cr2.MultiPartIdentifier.Count; x++)
        {
            if (x > 0) { ans += "."; }
            ans += cr2.MultiPartIdentifier[x].Value;
        }
    }
    return ans;
}
```

## SchemaToTableName()

This function takes a named table reference object (properties such as server name and schema) and returns a string containing the qualified table name. It takes two parameters: the table name object and a Boolean flag indicating whether or not to add delimiters around the table elements.

*Code Listing 57: SchemaToTableName()*

```
        /// <summary>
        /// Returns a full qualified table name from a table reference
        /// </summary>
        /// <param name="tbl"></param>
        /// <returns></returns>
        static public string SchemaToTableName(NamedTableReference tbl,
                        bool IncludeDelimters = true)
        {
            string TblName = "";
            string lDelim = "[";
            string rDelim = "]";
            if (!IncludeDelimters)
            {
                lDelim = "";
                rDelim = "";
            }

            if (tbl != null)
            {
                if (tbl.SchemaObject.ServerIdentifier != null)
```

```
                {
                    tblName += lDelim +
                        Tbl.SchemaObject.ServerIdentifier.Value+rDelim +
                  ".";
                }
                if (tbl.SchemaObject.DatabaseIdentifier != null)
                {
                    tblName += lDelim +
                        tbl.SchemaObject.DatabaseIdentifier.Value+rDelim +
                  ".";
                }
                if (tbl.SchemaObject.SchemaIdentifier != null)
                {
                    tblName += lDelim +
                        tbl.SchemaObject.SchemaIdentifier.Value + rDelim +
                  ".";
                }

                tblName += lDelim +
                        tbl.SchemaObject.BaseIdentifier.Value + rDelim;

            }
            return TblName;
        }
```

## Private methods

You can add your own methods (both public and private) as you work with ScriptDOM. For example, Code Listing 58 shows some private methods to convert an identifier or multi-part identifier to a string.

*Code Listing 58: MakeIdentifier methods*

```
#region PRIVATE_METHODS
private static string MakeIdentifier(MultiPartIdentifier obj)
{
    string ans = "";
    for (int x = 0; x < obj.Count; x++)
    {
        if (x > 0) { ans += "."; }
        ans += MakeIdentifier(obj.Identifiers[x]);
    }
    return ans;
}
private static string MakeIdentifier(Identifier obj)
{
```

```
    string ans = MakeIdentifier(obj.Value.ToString(), obj.QuoteType);
    return ans;
}
private static string MakeIdentifier(string objName,QuoteType quotetype =
QuoteType.NotQuoted)
{
    string leftDelim = "[";
    string rightDelim = "]";
    if (quotetype==QuoteType.NotQuoted)
    {
        leftDelim = "";
        rightDelim = "";
    }
    return leftDelim+objName+rightDelim;
}
#endregion
```

You can use the method with various parameters to create a string representation of an identifier found in your script.

## Summary

These functions can be handy with some of the common objects you'll encounter using the ScriptDOM library. Feel free to add your own as you work more with ScriptDOM.

# Chapter 15  Generate Script

Our newbie developer from Chapter 1 has written some stored procedures for us, but unfortunately, has not yet developed good code-formatting habits. His latest stored procedure code is shown in Code Listing 59.

*Code Listing 59: Newbie's get inactive customers*

```
CREATE proc dbo.InactiveCustomers
AS
begin
  DECLARE @oneYearAgo date
  set @oneYearAgo = DateAdd(year,-1,getDate());
        select cs.first as FirstName, cs.last as LastName,
 z.city,z.state,cs.zipCode
        FROM customer cs JOIN zipCodes z on z.zipCode=cs.zipCode
  WHERE cs.lastVisit > @oneYearAgo ORDER BY cs.lastVist
insert logTable (tableName,updatedBy,updatedDate)
values ('CUSTOMER',user_Name(),
getDate() )
END
```

In his defense, the code does work, but it is difficult to read and maintain. Fortunately, ScriptDOM comes to the rescue.

## SqlxxScriptGenerator (where xx is version #)

There are different versions of the Script Generator, depending on your SQL version. Figure 15 shows a sample WinForms application in which we've loaded the ugly code from the newbie.

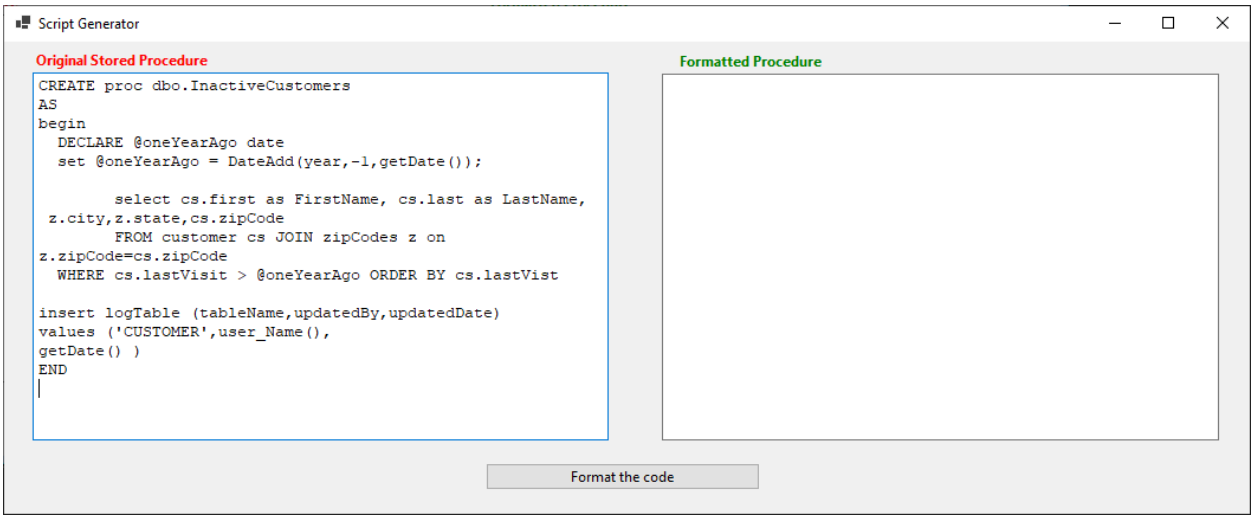


*Figure 15: WinForms application*

## Format the code

The **Format the code** button performs the magic, using ScriptDOM to format the code. Code Listing 60 shows the Format the code button code. It starts with the usual code snippet to create the abstract tree, but then calls a script generator method to format the new code.

*Code Listing 60: Format code*

```
        private void FormatCodeBtn_Click(object sender, EventArgs e)
        {
            var parser = new TSql160Parser(true, SqlEngineType.All);
            // Create list to hold any errors
            IList<ParseError>? errors = null;

            byte[] byteArray = Encoding.ASCII.GetBytes(sb.ToString());
            MemoryStream stream = new(byteArray);
            StreamReader rdr = new(stream);

            // Ask ScriptDOM to parse it
            TSqlScript tree = (TSqlScript)parser.Parse(rdr, out errors);
            Sql160ScriptGenerator generator = new Sql160ScriptGenerator();

            generator.GenerateScript(tree, out string srcCode);
            FormattedCode.Text = srcCode;
        }
```

Once ScriptDOM formats the code, we can load it into our application to see the result. Figure 16 shows the original and formatted code.

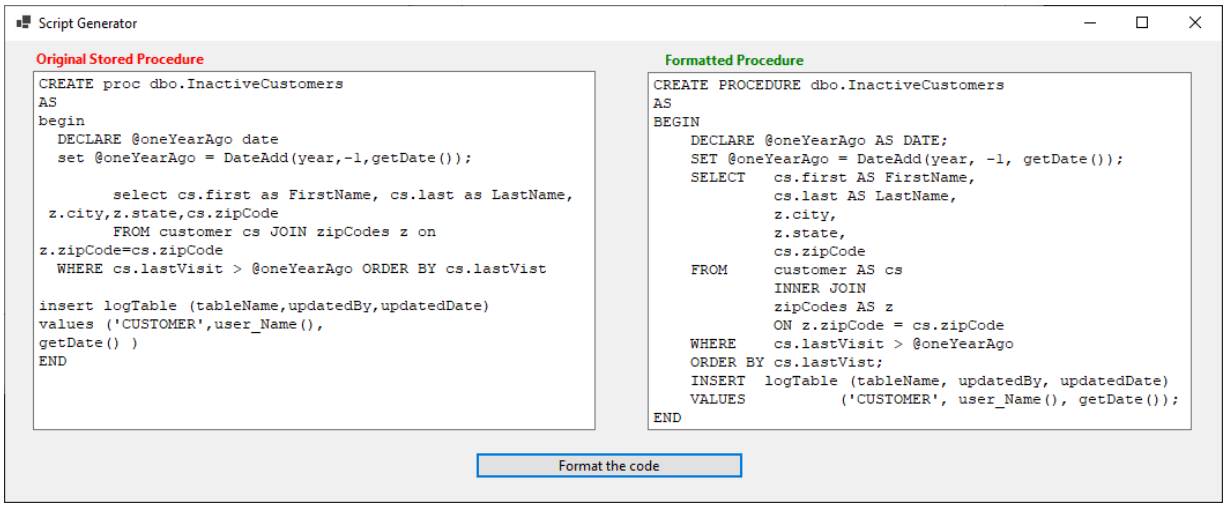


*Figure 16: Original and formatted code*

## Formatting options

There are a number of options you can specify to control how the code is formatted. You set these on the generator object before calling the **`GenerateScript`** method. For example, to specify lowercase keywords, you would use the following code snippet:

```
generator.Options.KeywordCasing = KeywordCasing.Lowercase;
```

Table 33 lists some of the different options you can set to form the code.

*Table 33: Code formatting options*

| Option | Type | Description |
|---|---|---|
| **`AlignClauseBodies`** | Boolean | **`FROM`**, **`WHERE`**, etc. are aligned |
| **`AlignSetClauseItem`** | Boolean | Align **`SET`** statements in **`UPDATE`** |
| **`AsKeywordOnOwnLine`** | Boolean | Should each **`AS`** be on its own line |
| **`IncludeSemicolons`** | Boolean | Include semicolons as line delimiters |
| **`KeywordCasing`** | Upper Case<br>Lower Case<br>CamelCase | Options to show how keywords are cased<br>CREATE, create, Create |
| **`NewLineBeforeFromClause`** | Boolean | Include a new line before the **`FROM`** clause |
| **`NewLineBeforeJoinClause`** | Boolean | Include a new line before a **`JOIN`** clause |
| **`SQLVersion`** | Enum List | Determine SQL version for script |

The complete list of options can be found here. You can customize the options to format the code to your company preferences.

 ***Note: Comments do not get copied to the formatted code.***

## Summary

If your code is pretty ugly, you can use ScriptDOM to tidy it up for you. You will likely need to add the comments back in, but this should at least allow you to get more consistent and readable SQL stored procedures.

# Chapter 16 SQL Versions

SQL Server has been around since June 1995 (version 6). Each version adds new functionality (and sometimes deprecates features as well). Each version of SQL after SQL 2000 has an associated ScriptDOM class, allowing you to ensure the code you are checking is using valid syntax and options for your SQL version.

***Note: SQL versions prior to SQL Server 2019 are no longer supported.***

## ScriptDOM parser versions

Table 34 lists the various SQL versions and the appropriate ScriptDOM parser to use.

*Table 34: ScriptDOM versions*

| SQL Version | Internal | Parser object |
|---|---|---|
| 2022 | 16 | TSql160Parser |
| 2019 | 15 | TSql150Parser |
| 2017 | 14 | TSql140Parser |
| 2016 | 13 | TSql130Parser |
| 2014 | 12 | TSql120Parser |
| 2012 | 11 | TSql110Parser |
| 2008 | 10 | TSql100Parser |
| 2005 | 9 | TSql90Parser |
| 2000 | 8 | TSql80Parser |

You should use the appropriate parser based on the SQL version. The various objects returned for each statement will only contain properties that were available for that version of SQL Server.

## Summary

Use the parser for your version of SQL Server to ensure your analysis doesn't miss features or accept code that is not valid for the SQL version.

# Chapter 17 Interacting with a Database

In each chapter of the ebook, we've embedded stored procedure code directly into the application source code. However, it is much more useful to pull stored procedures from the database, load them, and call whichever visitor patterns you want to use.

In this chapter, we will create a basic C# program to grab procedure source code, load it into the stream, and let ScriptDOM parse it. The code will display the results in a text window and allow you to save the text file for further analysis and review. You can create or combine the visitor methods you want to use to perform code analysis on all code within a database.

## Connecting to a server

To connect to a particular server, you will need a connection string with your credentials. Table 35 shows the elements of the connection string.

*Table 35: Connection string*

| Parameter | Meaning |
|---|---|
| **`Data Source`** | Name of the server |
| **`Initial Catalog`** | Database on the server to connect to |
| **`Integrated Security`** | True: Use Windows credentials<br>False: Provide user ID and password |
| **`User ID`** | If no integrated security, the user ID for SQL |
| **`Password`** | If no integrated security, the password for SQL |
| **`Multiple Active Result Sets`** | Set to **`TRUE`** to allow more than one result at a time |

The code in Code Listing 61 can be used to open the connection by using the connection string.

*Code Listing 61: Connect to SQL database*

```
using System.Data;
using System.Data.SqlClient;

private readonly SqlConnection theConnection = new SqlConnection();

private string SqlVersion;
```

```
        /// <summary>
        /// Open a connection
        /// </summary>
        /// <returns>OK or an error message</returns>
        public static string OpenConnection(string ConnectionString)
        {
            string isOK = "OK";
            // If connection is not opened, try to open it
            if (theConnection.State != ConnectionState.Open)
            {
                try
                {
                    theConnection.ConnectionString = ConnectionString;
                    theConnection.Open();
                }
                catch (Exception e)
                    {
                        isOK = e.Message;
                    }
                }
            }
            return isOK;
        }
```

If the code is successful, the connection will be made and **OK** will be returned. Otherwise, the return value will be the error message.

## Running a SQL query

Once you connect, you should get the SQL version so you can make sure you use the proper version of the ScriptDOM parser. Code Listing 62 shows code to run a SQL query and return a table object.

*Code Listing 62: Get table from SQL*

```
        public DataTable SQLGetTable(string theQuery)
        {
            using (SqlCommand theCommand = new SqlCommand(theQuery))
            {
                theCommand.Connection = theConnection;
                try
                {
                    DataTable ans_ = new DataTable("TBL");
                    SqlDataReader a = theCommand.ExecuteReader();
                    ans_.Load(a);
                    a.Close();
```

```
                return ans_;
            }
            catch (Exception e)
            {
                return null;
            }
        }
    }
```

We can pass the query **`SELECT @@version`** and get the first row and column if the function does not return a null table. Code Listing 63 shows an example call and transfers the value into the **`SqlVersion`** variable.

*Code Listing 63: Get SQL version*

```
            DataTable dt = SQLGetTable("SELECT @@version");
            if (dt !=null)
            {
                SqlLVersion = dt.Rows[0].ItemArray[0].ToString()
            }
```

We save the version to a global string so we can use it to determine the parser to use.

## Getting all procedure code

Using our SQL to table function, we can create a SQL query to get all of the stored procedures and return them as a C# table. Code Listing 64 shows this code.

*Code Listing 64: Get all stored procedures*

```
//============================================
// Get routines you need to process
//============================================

    string SQL = "SELECT r.ROUTINE_SCHEMA+'.'" +
                 "+r.ROUTINE_NAME as RtnName " +
                 "FROM INFORMATION_SCHEMA.ROUTINES r ";

    DataTable? dt = SQLManager.SQLGetTable(SQL);

    if (dt !=null)
       {
         // dt contains a list of procedures from the database
         // (see Code Listing 65 for actions)
       }
```

We can then use the **dt** C# table object to process all procedures in the database.

> ***Note: If you are interested in finding out more about the information schema views and other SQL metadata tables, be sure to check out my Succinctly® series ebook SQL Server Metadata Succinctly.***

## Looping through the code

Once you have the list of stored procedures, you can loop through that list and call the parser code of each item. We are going to use the **sp_HelpText** system procedure to get the actual procedure content.

Code Listing 65 will grab all the procedures and return a dictionary containing the routine name and the source code.

*Code Listing 65: Get procedure source code*

```
private Dictionary<string, StringBuilder> GetSourceCode(DataTable dt)
{
    List<string> RoutineNames = new List<string>();
    Dictionary<string, StringBuilder> srcCode = new Dictionary<string,
StringBuilder>();
    StringBuilder sb = new StringBuilder();
    // Get list of routines into a list for processing
    foreach (DataRow dr in dt.Rows)
    {
        RoutineNames.Add(dr["RtnName"].ToString() );
    }
    // Step through each routine
    foreach (string rtn in RoutineNames)
    {
        string[] parts_ = rtn.Split('|');
        string routineName = parts_[0];

        // Get the routine source code
        dt = SQLGetTable("EXEC sp_helpText '" + routineName + "';");
        if (dt != null && dt.Rows.Count > 0)
        {
            sb.Clear();
            string curLine = "";
            foreach (DataRow dr in dt.Rows)
            {
                curLine = dr[0].ToString();
                sb.Append(curLine);
            }
```

```
            StringBuilder newSB = new StringBuilder(sb.Length);
            newSB.Append(sb);
            srcCode.Add(routineName, newSB);
        }
    }
    return srcCode;
}
```

Once we have the dictionary built, we can loop through it and call whichever visitor methods we are interested in processing. Code Listing 66 shows the code to loop through the dictionary and process it.

*Code Listing 66: Process all routines*

```
private void ProcessCode(Dictionary<string, StringBuilder> procList)
{
    string SQL;
    IList<ParseError>? errors = null;

    foreach (var pb in procList)
    {
        SQL = pb.Value.ToString();
        byte[] byteArray = Encoding.ASCII.GetBytes(SQL);
        MemoryStream stream = new MemoryStream(byteArray);
        // convert stream to string
        StreamReader reader = new StreamReader(stream);
        TSql150Parser parser = new TSql150Parser(true);
        TSqlFragment tree = parser.Parse(reader, out errors);

        MyVisitor mv = new MyVisitor();
        tree.Accept(mv);
    }
}
```

You can write your own visitor class to process whichever statements you choose. Most of our examples write to the console window, but feel free to process however you'd like.

## Summary

Pulling the code from the database is simply a connection and a few queries. Once you pull the list of code, load it up and let ScriptDOM process it. In this ebook, we create various console errors, so you can identify the ones you are interested in and create a report of all stored procedures that have the issues.

# Chapter 18 Answers

This chapter explains the various issues from the sample procedures in Chapter 1. With the ScriptDOM library, we can write code to catch all of these conditions.

## Newbie stored procedure

Looking at the newbie's code in the following Listing, we found a couple of issues.

*Code Listing 67: Newbie's stored procedure*

```
CREATE procedure [FindUsersByLanguage](@languagecode varchar)
as
begin
      set nocount on
      declare @numPeopleFound int
      select * from [Person]
      where LanguageID is null or LanguageID <> @languagecode
end
```

First, the parameter **@languageCode** is a varchar data type, but without a length specified, the default size is 1 byte. The user might pass in a language code, such as EN-US, but only one character will be received in the code.

The **@numPeopleFound** variable is never used; while this isn't a bug, unused variables should be removed from code as a general guideline.

It is also good practice to include the schema name when creating or altering a procedure.

***Note:** `SELECT *` **should generally be avoided as well.***

## Report code

The user only needs a few fields, but the query is returning every field, including the very large image field. For many tables, this might not be noticeable, but the image field is likely to be very large, so bringing it back will slow down the query. Code Listing 68 shows the report code.

*Code Listing 68: Report code*

```
CREATE procedure [dbo].[PhoneList] (@whichparty varchar(12))
as
begin
```

```
      select * from dbo.voters
      where upper(party_affiliation) = @whichparty
end
```

**select * from dbo.voters**

The quick fix is the following:

Replace **select * from dbo.voters**

with **select FirstName, LastName, Birthday from dbo.voters**

***Tip: You should always only bring back the fields needed for the result.***

However, the code also should not need the **upper()** function, since most SQL installs are case-insensitive. However, any function in the column name will reduce the likelihood that any index can be used.

## Add new company

This code is problematic because it was working, but something in the database has changed. In this case, we found that a trigger was added to the company table, logging the database user who added the company. Code Listing 69 shows the original code.

*Code Listing 69: Add new company*

```
CREATE procedure [dbo].[InsertCompany] (@companyName varchar(50),@zipCode
varchar(10) )
as
begin
      DECLARE @companyID int        -- Record # of new company

      INSERT INTO dbo.Company (CompanyName,PostalCode)
      VALUES (@companyName,@zipcode)

      SET @companyID = @@IDENTITY

      UPDATE dbo.Company
      SET city = zp.City,StateCode = zp.StateCode
            FROM (select * from dbo.ZipCodes WHERE zipCode=@zipCode) zp
      WHERE CompanyID = @companyID
end
```

In this case, the **@@identity** function is causing the problem. This function returns the last identity key within the session. In this code, that will be the identity key from the log table, not the identity key from the company table.

The solution is to replace **@@identity** with **Scope_Identity()**. **Scope_Identity()** returns the last identity key with the session AND within the procedure scope. In this case, the **@companyID** will be set to the identity key of the company, not the log table.

## User report

The user report is tricky because, at first glance, nothing appears to be wrong, and the query looks very simple. Code Listing 70 has the report query.

*Code Listing 70: User report*

```
CREATE procedure [dbo].[UsersReport]
as
begin
      SELECT      sur.Email
                  ,u.UserName as 'UserName'
                  ,u.FirstName as 'First Name'
                  ,u.LastName as 'Last Name'
      FROM [StoreUser] sur
      JOIN [AspNetUsers] u ON u.Id = sur.id
End
```

The problem is not obvious until you look at the tables used. In the larger table, **AspNetUsers**, the **Id** column is a **nvarchar(256)** value, while in the smaller table, the **Id** column is a unique identifier. This causes a data type mismatch, so SQL needs to convert the **AspNetUsers Id** column to a unique identifier to perform the **JOIN** operation. This is called an *implicit conversion.*

Implicit conversions prevent indexes from being used and will substantially slow down a query. In this case, the simple solution is to convert the unique identifier explicitly to the **nvarchar(256)**, which will allow SQL to join using indexes.

So, let's change the **JOIN** from:

```
JOIN [AspNetUsers] u ON u.Id = sur.id
```

to:

```
JOIN [AspNetUsers] u ON u.Id = cast(sur.id as nvarchar(256))
```

If you run the query and view the execution plan, it will point out implicit conversions. Be sure to address them to gain substantial improvements in performance.

***Note: Be sure to put the `cast` on the smaller table.***

## Payroll problem

The payroll code from Chapter 3 uses the following logic to compute the pay period:

```
DECLARE @15DaysAgo Date = DATEADD(day,-15,@payrollDate )
```

When we run a payroll on the 15th of the month, the date range includes the last date of the prior month. This could cause an extra day to be included when summing up the daily pay. Similarly, if the month has 30 days, then the 15th date will also be included when computing the amount to pay.

Since SQL 2012, there has been an **EOMONTH()** function, which returns the last date of the month. You can use the following snippet to find the first date of the month:

```
SELECT DATEADD(DAY, 1, EOMONTH(GETDATE(),-1)) AS 'FirstDayOfMonth'
```

This basically says: get the end of the prior month and add one day to get the start date. So, when payroll is the 15th, use this formula. When payroll is at the end of the month, you can use the **DATEFROMPARTS** function, as shown in the following snippet:

```
SELECT DATEFROMPARTS(year(@payrollDate),month(@payrollDate), 16)
```

## Summary

While we used ScriptDOM to automate the various issues found, this chapter should throw some additional light on things to review when you are looking at stored procedure code in SQL.

# Chapter 19 Conclusion

Imagine you are tasked with determining all tables referenced in a **SELECT** statement. How easy or difficult would it be to handle all options available to the statement? For example:

- `SELECT * from Customer`
- `SELECT first_name,Last_name FROM [dbo].[Customer]`
- `SELECT first_name, user_email FROM UserTable ut JOIN StoreUser s ON ut.id=s.id`
- `SELECT * FROM (SELECT stateName,count(*) from StateTable) X`

Parsing SQL code is a tough task, but thankfully, ScriptDOM is a great tool for the job. It is a little-known gem in the Microsoft .NET Framework. Although it has a limited scope (SQL parser and formatter), it is very good at those jobs. Hopefully, this ebook has given you some ideas about how to analyze SQL stored procedure code.

***Note: While ScriptDOM is specifically for Microsoft SQL, you can probably use it for other SQL dialects as well, particularly if you avoid any product-specific features.***

## More resources

Here are the Microsoft resources for the ScriptDOM library:

- NuGet package
- API documentation

### GitHub open source

The ScriptDOM source code is licensed under the MIT license agreement. The code is available on GitHub: https://github.com/microsoft/SqlScriptDOM/tree/main.

## Recommended links

- ScriptDOM samples: https://github.com/arvindshmicrosoft/SQLScriptDomSamples
- ScriptDOM articles on SQL Server Central: https://www.sqlservercentral.com/search/SCRIPTDOM

## General links

If you are interested in some theory, you can check out the following links:

- Visitor pattern: Behavior design pattern for the visitor
- Abstract syntax tree: The tree structure used by ScriptDOM

## Summary

Using ScriptDOM and some C# coding, you can analyze stored procedures, looking for errors, lack of standards, performance suggestions, etc. It is just one of those mighty hidden gems buried in the .NET Framework.