☰ Menu

Sql Quantum Leap

Stuff about computers and programming, mostly related to Microsoft SQL Server

Prevent Full Script Execution (Understanding and Using PARSEONLY and NOEXEC)

Posted on December 28th, 2018

(last updated: 2019-01-29 @ 22:30 EST / 2019-01-30 @ 03:30 UTC)

There are times when I am working on a SQL script that really shouldn't be executed all at once. Sometimes it's a series of examples / demos for a presentation or forum answer. Other times it's just a temporary need while I'm in the process of creating a complex script, but once the script is completed and tested then it should run all at

ran until completion or until I was able to cancel it (if it ran long enough for me to have time to understand what was happening *and* hit the "cancel" button).

So I needed some way of ensuring that a script would not execute if no section was highlighted. The following is what I ended up using — `PARSEONLY` — and something similar that I learned along the way — `NOEXEC` — that might prove useful in other situations.

Background

`PARSEONLY` and `NOEXEC` are both session-level settings, enabled / disabled via `SET` statements (just like `NOCOUNT`). In order to understand the effect that each one has, we need to take a quick look at how SQL Server handles ad hoc requests (stored procedures executed as RPC's – Remote Procedure Calls – are handled slightly differently, and are not the focus of this post since we are concerned with ad hoc scripts).

The two main aspects of ad hoc request processing that need to be understood in order to fully understand how these two session settings work are:

1. SQL Server receives, and processes, one query *batch* at a time. A query batch is one or more T-SQL statements. Multiple batches can be submitted through a client application[1] separated by a "batch separator". The batch separator used by SSMS and SQLCMD (and probably others) is `GO` on a line by itself (well, technically you can include an optional INT value after it to repeat the batch above it that many times). This value is configurable, though in practice I have never seen anyone ever use something other than the default `GO` .

   If there are multiple batches, each one is submitted to SQL Server when the client app reaches a batch separator. Batches are completely separate "requests" (i.e. sys.dm_exec_requests). This is why local variables are not known between batches, and why you need session-level constructs such as `CONTEXT_INFO`, `session_context`, and temporary objects to pass information from one batch to the

1. The **Parse** phase only parses the syntax of the statements. This means making sure that the statements are valid T-SQL statements, that variables have been declared, that regular (i.e. non-delimited) identifiers follow the rules for regular identifiers, etc.
2. The **Compile** phase checks permissions, makes sure that the objects exist, finds / applies optimizations, and comes up with an execution plan.
3. The **Execute** phase executes the statement(s) in the query batch and returns any messages and/or results.

Again, and assuming that there are no errors, each query batch will go through all three phases before the next query batch is processed.

SQLCMD / SQLCMD mode

Even though it is not entirely on topic for this post, since the following information does relate to how scripts and batches are processed, I will point out that SQLCMD commands and variables (available in `SQLCMD.exe` and "SQLCMD mode" within SSMS) are processed:

- for the current batch only! If there are additional batches that contain SQLCMD commands and/or variables, those will be processed when the client app gets to that batch
- *before* the **Parse** phase. This is because SQLCMD commands and variables are processed only by the client app and are unknown to SQL Server (just like the `GO` batch separator)
- allowing SQLCMD variables to retain their values over the entire execution, across multiple batches (unlike T-SQL variables)

PARSEONLY

The `PARSEONLY` setting will prevent the processing from entering the **Compile** phase (and if it is not obvious, the **Execute** phase will also be skipped). Because this setting

1. this can be applied anywhere in a batch
2. it cannot be applied conditionally since `IF` statements are evaluated in the **Execute** phase
3. if you enable and then disable this setting in the same batch, then the batch will execute as it normally would since the disabling of the option is occurring in the **Parse** phase, allowing the next two phases to proceed

Basics

To better illustrate this behavior, let's look at five simple examples.

The following example shows the expected error when a query references a column that does not exist in a table that does exist:

```
-- Example 1:

DECLARE @T TABLE (Col1 INT);
SELECT [Col2] FROM @T;
/*
Msg 207, Level 16, State 1, Line XXXXX
Invalid column name 'Col2'.
*/
GO
```

The following example shows that enabling and disabling `PARSEONLY` in the same query batch has no effect:

```
-- Example 2:
SET PARSEONLY ON;
DECLARE @T TABLE (Col1 INT);
SELECT [Col2] FROM @T;
SET PARSEONLY OFF;
/*
Msg 207, Level 16, State 1, Line XXXXX
Invalid column name 'Col2'.
*/
GO
```

The following example shows both that:

1. you need to disable the option in a separate batch, and
2. where in a batch it is enabled does not matter since it happens in the **Parse** phase


```
GO
```

```
SET PARSEONLY OFF;
/*
Commands completed successfully.
*/
GO
```

The following example shows that even with this option enabled, the batch will still be parsed for proper syntax:

```
-- Example 4:
SET PARSEONLY ON;
DECLARE @T TABLE (Col1 INT);
SELECT [Col2] FROM @Tt;
GO
SET PARSEONLY OFF;
/*
Msg 1087, Level 15, State 2, Line XXXXX
Must declare the table variable "@Tt".
*/
GO
```

The following example shows that when this option is enabled, it is only proper syntax that is being parsed and not things like object existence, data type usage, etc:

```
-- Example 5:
SET PARSEONLY ON;
SELECT [GhostColumn] FROM sys.objects;
DECLARE @T INT = NEWID();
GO
SET PARSEONLY OFF;
/*
Commands completed successfully.
*/
```

With those concepts in mind, it should be easier to understand what is happening in the following examples:

```
GO
PRINT 1;

SET PARSEONLY ON;

PRINT 2;

SET PARSEONLY OFF;

PRINT 3;
GO
/*
```

```
-- Dynamic SQL does not affect the calling / outer context
EXEC(N'SET PARSEONLY ON;');
PRINT 4;
GO

PRINT 5;

IF (1 = 0)
BEGIN
  SET PARSEONLY ON; -- takes effect in "parse" phase
END;

PRINT 6;

-- Dynamic SQL does not affect the calling / outer context
EXEC(N'PRINT 7; SET PARSEONLY OFF; PRINT 8;');

PRINT 9;

GO
PRINT 10;
SET PARSEONLY OFF; -- takes effect in "parse" phase
PRINT 11;
GO
```

The batch above returns the following:

```
4
10
11
```

Template

Here is the template that I use in my scripts. The `PRINT` statements at the top and bottom only execute if the entire script is run, in which case nothing else is executed.

```
PRINT 'This script is not meant to execute all at once!';
PRINT 'Please highlight and execute each section individually.';
GO
SET PARSEONLY ON;
GO


PRINT 'Doin'' sumthin'';


GO
```

NOEXEC

The `NOEXEC` setting will only prevent the processing from entering the **Execute** phase, but only for the statements that follow it. Because this setting is handled in the **Execute** phase:

1. it needs to be enabled prior to any statements that you want to prevent the execution of
2. it *can* be applied conditionally since `IF` statements are also evaluated in the **Execute** phase
3. you can enable and then disable this setting in the same batch

Basics

This option works in the **Execute** phase and so requires less explanation since it operates like most other statements.

The following example shows that while the statements in this batch did not produce any parse errors, they certainly don't compile:

```
SET NOEXEC ON;
PRINT 'This will not print';
GO
SELECT [GhostColumn] FROM sys.objects;
DECLARE @T INT = NEWID();
GO
SET NOEXEC OFF;
/*
Msg 207, Level 16, State 1, Line XXXXX
Invalid column name 'GhostColumn'.
Msg 206, Level 16, State 2, Line XXXXX
Operand type clash: uniqueidentifier is incompatible with int
*/
```

The following query shows that this option does indeed take effect in the **Execute** phase, and can be used in the same batch:

```
GO
```

```
SET NOEXEC OFF;

PRINT 14;
GO
/*
12
14
*/
```

The following example illustrates how this option can be used conditionally:

```
-- Dynamic SQL does not affect the calling / outer context
EXEC(N'SET NOEXEC ON;');
PRINT 15;
GO
DECLARE @StopExecuting BIT;
SET @StopExecuting = 1;

PRINT 16;

IF (@StopExecuting = 1)
BEGIN
  SET NOEXEC ON; -- takes effect in "execution" phase
END;

PRINT 17;

-- Dynamic SQL does not affect the calling / outer context
EXEC(N'PRINT 18; SET NOEXEC OFF; PRINT 19;');

PRINT 20;

--GO
SET NOEXEC OFF; -- takes effect in "execution" phase
PRINT 21;
GO
```

If `@StopExecuting = 1`, the output will be:

```
15
16
21
```

However, if `@StopExecuting = 0`, the output will be:

```
15
16
17
```

Template

```
DECLARE @RequirementsMet BIT;


PRINT 'Check and prepare things...';
SET @RequirementsMet = 0;


IF (@RequirementsMet = 0)
BEGIN;
    RAISERROR('error message...', 10, 1) WITH NOWAIT;
    SET NOEXEC ON;
END;


-- Stuff that should not execute if conditions are not met
PRINT 'This will not execute if NOEXEC is ON!';


-- Place at end of script to make sure NOEXEC is not left ON.
-- This can be in the same query batch, or in a different batch.
SET NOEXEC OFF;
GO
```

PARSEONLY and NOEXEC Together

If you combine these two options, it will only be the effect of `PARSEONLY` that you see because processing will not continue to the **Compile** phase. For example:

```
SET NOEXEC ON; -- affects statements that follow

SELECT x FROM sys.objects; -- compile error (no error if PARSEONLY i

SET PARSEONLY ON; -- affects entire batch, regardless of location
GO

SELECT 1 / 0; -- execution error (no error if PARSEONLY or NOEXEC is
GO

SET PARSEONLY OFF;
SET NOEXEC OFF;
SELECT 1;

-- 1
```

Executing the T-SQL shown above will not result in any errors. This is because there are

continues in the second batch. There are still no errors because `PARSEONLY` is still enabled.

If you comment out `SET PARSEONLY ON;` and then execute, you will get the following errors:

```
Msg 207, Level 16, State 1, Line XXXXX
Invalid column name 'x'.

Msg 8134, Level 16, State 1, Line XXXXX
Divide by zero error encountered.
```

Why do we get the "Divide by zero" error? Because `NOEXEC` was never enabled. The "Invalid column name" error aborted the batch in the **Compile** phase, so the `SET NOEXEC ON;` statement never executed. Processing continued to the second batch with neither option enabled.

If you also comment out the `SELECT x...` line and then execute, there will once again be no errors. This is due to `NOEXEC` being enabled this time.

What all of this means is: there's no purpose in combining these two options / settings. Use one or the other.

Conclusion

The setting of these two options has the following effect on processing:

PARSE ONLY	NO EXEC	Execution Phases	Notes
ON		Parse	
ON	ON	Parse	same as PARSEONLY by

PARSE ONLY	NO EXEC	Execution Phases	Notes
		Parse ➔ Compile (and Optimize) ➔ Execute	(default)

For my purposes I prefer the `PARSEONLY` option since a) I don't need to handle anything conditionally, and b) the only errors that can occur are parse errors. This makes it more likely that the only error I see is the one telling me that I accidentally executed the entire script (as opposed to one or more potential compilation errors that sometimes make me think that the `NOEXEC` option didn't work and that some statements actually executed).

Database Project deployment scripts generated by SSDT (usually used within Visual Studio) use the `NOEXEC` option since they do have conditional processing to ensure that requirements are met.

DOCUMENTATION

- SET PARSEONLY
- SET NOEXEC
- SQL Server Management Studio (SSMS) (GUI)
- sqlcmd Utility (command-line utility)
- Azure Data Studio (GUI ; originally named "SQL Operations Studio")
- osql Utility (command-line utility ; officially deprecated / do not use)
- SimpleSqlExec (command-line utility ; open source / C# / written by me)

1. SQL Server Management Studio / SSMS and SQLCMD.exe are the most common SQL Server client applications, but there are others. OSQL, which has been deprecated for a while now and should not be used, predates SQLCMD, which is the current default command-line utility. Another command-line utility option is SimpleSqlExec, which is a .NET-based open source project that I wrote (and host on GitHub). Azure Data Studio is a new-ish GUI from Microsoft that, unlike SSMS, also

Like this:

Loading...

Posted in Programming

Tagged T-SQL

Previous Post

SQLCLR vs SQL Server 2017, Part 9: Does PERMISSION_SET Still Matter, or is Everything Now UNSAFE?

Next Post

Beware! Beware of Unintended Changes When Altering Columns!

1 thought on "Prevent Full Script Execution (Understanding and Using PARSEONLY and NOEXEC)"

Preventing Execution With PARSEONLY And NOEXEC – Curated SQL

December 31st, 2018 Reply

[...] Solomon Rutzky shows us a way to prevent accidental full script execution: [...]

Loading...

Leave a Reply

Recent Posts

SSMS: Prevent Copy and Paste of Text in “Messages” tab (Cruel Joke #2) May 22nd, 2020

Presenting at “Data Architecture Day” this Saturday (May 16th) at 4:10 PM EDT May 14th, 2020

sys.xp_delete_files and ‘allow filesystem enumeration’: two new undocumented items in SQL Server 2019 January 26th, 2020

Actual Difference Between EXISTS(SELECT 1 ...), EXISTS(SELECT * ...), and EXISTS(SELECT column ...) January 24th, 2020

How Many Bytes Per Character in SQL Server: a Completely Complete Guide November 22nd, 2019

Categories

Announcements

Collation

General

Programming

SQLCLR vs. SQL Server 2017

Archives

May 2020

June 2019
May 2019
April 2019
March 2019
February 2019
January 2019
December 2018
October 2018
September 2018
August 2018
June 2018
April 2018
March 2018
February 2018
January 2018
December 2017
November 2017
October 2017
September 2017
August 2017
July 2017
May 2017

Follow Blog via Email

Enter your email address to follow this blog and receive notifications of new posts by email.

Email Address

RSS - Posts

RSS - Comments