Optimization and Recompilation

Kimberly L. Tripp
SQLskills.com
@KimberlyLTripp

pluralsight
hardcore dev and IT training

When Does a Procedure Get Optimized?

- **Only when it is executed (and there's no plan already in cache that meets the same execution criteria / conditions)**
- **Conditional logic**
  - Useful to breakdown different potential code paths
  - Does not optimize based on execution path
    - All statements that ***can*** *be* optimized, ***will*** *be* optimized
    - Based on the parameter values used at the time the plan is created (prone to parameter sniffing problems)
- **Modularization**
  - Supports multiple scenarios
    - Different execution paths with needed inline recompilation options
    - Different execution paths with sub-procedures that have consistent / stable plans
  - Cannot track executions of procedures or sub-procedures that are executed WITH RECOMPILE
  - Cannot push down WITH RECOMPILE to sub-procedures

Server-Wide: OPTIMIZE FOR UNKNOWN

- **Introduced / available in:**
  - SQL Server 2008 R2 Cumulative Update 2
  - SQL Server 2008 Service Pack 1 (SP1) Cumulative Update 7
  - SQL Server 2005 Service Pack 3 (SP3) Cumulative Update 9
- **Enabled with trace flag 4136**
- **Disables parameter-sniffing server-wide and all queries use the density_vector (for averages) instead of the histogram (with specific parameter values)**
- **If you have a tremendous amount of parameter sniffing problems this is something to consider**
- **For more information, see KB article 980653**

The Checkered Past of OPTION (RECOMPILE)

- **SQL Server 2008 CU4 and SQL Server 2008 R2 CU1 fixed:**
  - KB article 968693 FIX: A query that uses parameters and the RECOMPILE option returns incorrect results when you run the query in multiple connections concurrently in SQL Server 2008
- **SQL Server 2008 R2 SP2 fixed:**
  - Problems where special performance-related features that should have been accessible with OPTION (RECOMPILE) were not
- **If I ever run into problems with OPTION (RECOMPILE), I always consider using a dynamic string instead**
  - This always works!

Dynamic String Execution

```
CREATE PROCEDURE DSEtest ( @objname SYSNAME ) AS
DECLARE @Str NVARCHAR(4000);
SELECT @Str = N'SELECT TOP 10 * FROM ' + QUOTENAME(@objname);
EXEC (@Str);
```

- **String is undefined and unknown until runtime**
- **Statement acts as an ad hoc statement**
  - Depending on the setting for optimize for ad hoc workloads:
    - OFF: the statement and its plan are BOTH immediately placed in the cache and textual matches will use the plan
    - ON: on first execution only the statement is placed in cache and on second execution then the plan is also placed in cache
  - **If it's safe** then it is parameterized, placed in cache, and will be reused
  - **If it's unsafe** then only textual matches will have a plan (tied to optimize for ad hoc workloads) and all other executions will get their own plan

Multi-Purpose Procedures

- **Stored procedures with *n* parameters where any combination of these parameters can be supplied**
- **Often the code looks like this:**

```
WHERE (column1 = @variable1 OR @variable1 IS NULL)
 AND (column2 = @variable2 OR @variable2 IS NULL)
 ... AND (columnN = @variableN OR @variableN IS NULL)
```

- **These are very difficult for the optimizer to "generalize" and what often happens is a generally bad plan**
- **How do you fix it?**
  - Concatenate ONLY when the variable is NOT NULL
  - Determine the stability of different parameter combinations
    - For parameters that stabilize the plan – use **sp_executesql** normally
    - For parameters that create different optimal plans for different values – use **sp_executesql** with a statement that has **OPTION (RECOMPILE)**

Building Strings Dynamically and Caching

- **EXEC (@String) = ad hoc statements**
  - Act exactly as an ad hoc statement does
  - Can be cached when there are no parameters: subsequent executions must be an exact textual match (case-sensitive regardless of db setting and spaces)
  - Can be parameterized and deemed safe
    - Subsequent executions will use the plan in cache
  - Can be parameterized and deemed unsafe (MOST LIKELY)
    - Create ad hoc plan cache bloat
- **sp_executesql = forced statement caching**
  - The first execution will be "sniffed" and the plan will be placed in cache for subsequent executions (which is why we ONLY use this for stable plans)
  - Doesn't bloat the ad hoc plan cache
    - Using OPTION (RECOMPILE) on a statement makes it NOT show up (harder to troubleshoot as you won't see all of the executions)

Patterns and Practices in Statement Recompilation

- **Build only the statement you REALLY want to execute**
- **Combine conditional logic, modularization, and inline statement recompilation options to get the best performance**
  - OPTION (RECOMPILE) *OR* dynamic string execution
  - OPTION (OPTIMIZE FOR ...) *OR* OPTION (OPTIMIZE FOR UNKNOWN)
- **Best combination**
  - PRO: reduce recompiles and increase cache reuse where stability allows it (using **sp_executesql**)
  - PRO: reduce performance problems from executing plans optimized for different parameters (using a recompilation strategy)
  - CON: it takes time to test and analyze the best combination
- **So, where do you start?**
  - Your most expensive / problematic procedures first!

Summary: Stored Procedure Pitfalls/Performance

- Stored procedures and sp_executesql have the same potential for executing a bad plan but stored procedures have more options for *centralized* control
- Forcing a recompile can be warranted/justified
- Always recompile the smallest amount possible!