SQL Server: Optimizing Ad Hoc Statement Performance

Module 4: Statement Caching

Kimberly L. Tripp
Kimberly@SQLskills.com
http://www.SQLskills.com/blogs/Kimberly

pluralsight
hardcore developer training

Course Overview

- **Statement execution methods**
- **Estimates and selectivity**
- **Statement caching**
  - What affects ad hoc statement behavior?
  - Default ad hoc statement behavior
  - Ad hoc statement textual matching
  - Ad hoc statements – safe vs. unsafe
  - Ad hoc statement caching
  - Analyzing the plan cache
  - Changing ad hoc statement behavior
- **Plan cache pollution**
- **Statement execution summary**

What Affects Ad Hoc Statement Behavior?

- **Default behavior for ad hoc statements affected by server and database settings**
- **In SQL Server 2005, behavior defined solely by the database option/setting:**
  - Database option: *PARAMETERIZATION*
    - Default = *SIMPLE* (generally, recommended)
    - Optionally, *FORCED*
- **In SQL Server 2008 onward, an additional server setting affects ad hoc statement behavior:**
  - Server configuration setting for "*optimize for ad hoc workloads*"
    - Default = *OFF*
    - Optionally, *ON* (generally, recommended)

Default Ad Hoc Statement Behavior

- **On first execution (or, when SQL Server doesn't already have a plan in cache for a statement) then:**
  - Generate a plan, use it, and place it in the cache for future use
    - This is the cached plan type of "*Adhoc*" from *sys.dm_exec_cached_plans* (*objtype*) column
  - Analyze the statement to determine if it's "safe"
    - If it's safe – generate a plan, parameterize it, keep it in cache for subsequent executions $\Rightarrow$ called auto-parameterization
      - This is the cached plan type of "*Prepared*" from *sys.dm_exec_cached_plans* (*objtype*)
    - If it's unsafe, only the ad hoc plan will be placed in the cache
- **On subsequent executions, every statement checks to see if there's a match of the statement in the plan cache**
    - If there's an <u>**exact textual match**</u> of the statement, use that plan
    - If there's a <u>**parameterized plan**</u>, use that plan

Ad Hoc Statement Textual Matching

- **Must be exact in every way**
  - No spaces, tabs, or hard-returns
  - No case differences (even when the database is not case-sensitive)
- **All of these statements have a different textual plan in cache**

```
select * from member
select * from dbo.member
select  *    from member
select *    from    member
```

- **Each has a different *sys.dm_exec_sql_text (sql_handle)***
- **Each has a different *sys.dm_exec_query_plan (plan_handle)***

NOTE: Every statement above has the SAME *query_hash* (and, the same *query_plan_hash*)

Ad Hoc Statements – Safe vs. Unsafe

- **Statements are unsafe when the statement :**
  - Uses an *IN* clause
  - Has more than one table in the *FROM* clause
  - Uses expressions joined by *OR* in a *WHERE* clause
  - When a *SELECT* query contains a sub-query
- **OK, have I lost you yet? It's VERY restrictive (see whitepaper: *Plan Caching in SQL Server* 2008, Appendix A for a complete list)**
  - Available from http://bit.ly/zt8bw
- **A safe statement has a benefit – auto parameterization**
  - Idea: automatically parameterize search arguments and place the statement in cache for future re-use of the plan (saves CPU but not plan cache size)
  - Needs to be a VERY straightforward statement/plan
  - Parameters do not change plan choice
  - Doesn't happen very often and NOT something I would rely on and/or get excited about

Ad Hoc Statement Caching

- **Take the following "unsafe" query (identical, except for the SARG):**

```
SELECT …  WHERE [m].[lastname] = 'Tripp'
SELECT …  WHERE [m].[lastname] = 'Tripps'
SELECT …  WHERE [m].[lastname] = 'Tripped'
SELECT …  WHERE [m].[lastname] = 'Falls'
```

- **Each plan takes 16,384 bytes (or roughly 16KB)**
- **This "query class" is harder to track because each is listed in the cache**
- **All have the same *query_hash* but not necessarily the same *query_plan_hash***
- **Query *sys.dm_exec_query_stats* for *query_hash* and aggregate over *query_hash*, *query_plan_hash* to see how many plans a particular query might have in the cache**
- **If there's only one *query_plan_hash*, the statement may be stable**
  - However, this is completely index and data dependent

Verifying Plans in Cache NOW

- **Use *syscacheobjects* (in SQL Server 2000 and SQL Server 2005)**
  - *SELECT * FROM master.dbo.syscacheobjects*
  - This lists plans in cache as well as parameterized plans
  - *usecounts* and *refcounts* are interesting in terms of frequency of execution
  - *pagesused* gives insight into the size of the plan
- **Use DMVs to see the same and more in SQL Server 2005 onward**
  - DMV: *sys.dm_exec_query_stats* – gives much of what you see from *syscacheobjects*
  - DMF: *sys.dm_exec_sql_text (sql_handle)* – returns the actual text but only when needed (if just query stats then it's faster not to include text)
  - DMF: *sys.dm_exec_query_plan (plan_handle)* – can give you the XML Showplan as well and there's no prior equivalent
- **SQL Server 2008 added *query_hash* and *query_plan_hash* to *sys.dm_exec_query_stats***

Analyzing the Plan Cache

- **Every statement goes into the ad hoc plan cache for exact textual matching**
  - Review *sys.dm_exec_cached_plans*
- **SQL Server 2008 added *query_hash* and *query_plan_hash* to *sys.dm_exec_query_stats***
  - SQL templatizes the parameters – similar to *sp_get_query_template*
  - Aggregate by *query_hash* to find similar queries
  - Aggregate by *query_hash*, *query_plan_hash* to find similar queries and their plans
    - Queries that have only one *query_plan_hash* are STABLE
    - Queries that have more than one (sometimes dozens) are UNSTABLE
- **Check out Books Online topic "Finding and Tuning Similar Queries by Using Query and Query Plan Hashes" at http://bit.ly/QfUmRY**

The “Cumulative Effect” of Queries

```
SELECT [Query Hash] = [qs2].[query_hash]
, [Query Plan Hash] = [qs2].[query_plan_hash]
, [Avg CPU Time] = SUM ([qs2].[total_worker_time]) /
     SUM ([qs2].[execution_count])
, [Example Statement Text] = MIN ([qs2].[statement_text])
FROM (SELECT [qs].*, [statement_text] = SUBSTRING ([st].[text],
        ([qs].[statement_start_offset] / 2) + 1
        , ((CASE [statement_end_offset]
          WHEN - 1 THEN DATALENGTH ([st].[text])
          ELSE [qs].[statement_end_offset] END
             - [qs].[statement_start_offset]) / 2) + 1)
    FROM [sys].[dm_exec_query_stats] AS [qs]
    CROSS APPLY [sys].[dm_exec_sql_text]
        ([qs].[sql_handle]) AS [st]) AS [qs2]
GROUP BY [qs2].[query_hash], [qs2].[query_plan_hash]
ORDER BY [Avg CPU Time] DESC;
```

Changing Ad Hoc Statement Behavior (1)

- **Server setting: *optimize for ad hoc workloads***
  - On first execution, only the *query_hash* will go into cache
  - On second execution (if), the plan will be placed in cache
- **Significantly reduces the amount of cache allocated**
  - Query plans are on 8KB boundaries; plans can be quite large
    - Minimum plan size is 8KB and some plans can be MB in size
  - Plans for ad hoc statements that only execute once have only a few hundred bytes (just the statement/*query_hash*) in cache (rather than the ad hoc plan)
  - Does not eliminate plan cache pollution but slows it down
- **Generally, highly recommended**

Changing Ad Hoc Statement Behavior (2)

- **Database option: parameterization *FORCED***
  - More statements are forced to be cached on first execution (using parameter sniffing)
    - **Pro:** if you have stable plans (only one *query_plan_hash* for a *query_hash* that has lots of executions) from a lot of ad hoc clients then this might help to reduce CPU
    - **Con:** If you have some statements that really aren't safe, you could end up executing bad plans (PSP)
    - Recommendation: BEFORE turning this on, investigate plan stability by using *query_hash* and *query_plan_hash*
  - Not all statements are cached/parameterized
    - A *WHERE* clause condition defined by *LIKE* is not parameterized
    - *INSERT EXEC*
    - See Books Online topic "Forced Parameterization" for more information at http://bit.ly/1guvurN
  - Some features cannot be leveraged or may generate suboptimal plans in *FORCED* parameterization:
    - Filtered indexes and filtered statistics
    - Indexed views and indexes on computed columns
    - Partitioned tables and partitioned views
- **Generally, not recommended but highly useful when appropriate**

Multiple Plans (Tipping/Covering)

- **High-priority queries and queries that are executed often need to reduce their cumulative effect on the server**
- **By covering a query you can reduce the number of possible plans that exist for a query**
  - Reduces I/Os required to access the data the query needs
  - Reducing [often, drastically] I/Os translates into:
    - Fewer pages in cache results in better cache efficiency; less time
    - Important note: It's NOT just about I/Os – there are cases where plans with fewer I/Os are actually MORE expensive because of other operations (like sorts, temp tables, etc.) so be careful not to get too wrapped up in just I/Os. Review the showplan and statistics time as well!)
- **In some cases when you cover a query and create a stable plan, SQL Server might recognize it as safe (and parameterize/cache it)**
- **In other cases, even if all of your plans are stable, SQL Server might not recognize them as safe**
  - It's then when you might consider FORCED parameterization to save CPU

Summary: Statement Caching

- **Ad hoc statements**
  - Simple parameterization – almost all statements will be compiled just for that execution; they will not be parameterized/saved (see rules for parameterization in whitepaper)
  - Forced parameterization – most statements will be parameterized/saved (you'll see this in decreased CPU/compilations and potential for parameter-sniffing problems)
- ***sp_executesql*** **(or, prepared statements)**
  - A fantastic way to reduce the CPU/cache overhead that ad hoc has, but should only be used when a plan is stable and consistent
  - This is a better way to force a statement into cache as opposed to using forced parameterization database-wide
- **Stored procedures**
  - Can be "sniffed" but there are exceptions to what SQL Server will store in their plans