

## SQLskills Immersion Event IE2: Performance Tuning

### Module 9: Plan Cache & Index Analysis

Kimberly L. Tripp  
Kimberly@SQLskills.com



#### Overview

- Part I: Understanding/analyzing plan cache
- Part II: What's in the plan cache?
- Part III: Understanding/optimizing stored procedures
- Part IV: A cautionary tale about scalar functions
- Part V: Index analysis



2

© SQLskills, All rights reserved.  
<http://www.SQLskills.com>



## Overview

- **Part I: Understanding/Analyzing Plan Cache**
  - Statement execution
    - sp\_executesql
    - Dynamic string execution
  - Query fingerprints ⇒ the cumulative effect
  - Reducing Plan Cache Pollution
- **Part II: What's in the Plan Cache?**
- **Part III: Understanding/Optimizing Stored Procedures**
- **Part IV: A Cautionary Tale about Scalar Functions**
- **Part V: Index Analysis**

## Plan Cache Usage/Limits

- The “plan cache” (a.k.a. “procedure cache”)
- Uses “stolen” pages from the buffer pool (data pages)
- View cached plans: `sys.dm_exec_cached_plans`
- **Plan cache memory limits:**
  - SQL Server 2008+ and SQL Server 2005 SP2
    - 75% of visible target memory from 0-4GB
    - + **10%** of visible target memory from 4Gb-64GB
    - + **5%** of visible target memory > 64GB
  - SQL Server 2005 RTM and SQL Server 2005 SP1
    - 75% of visible target memory from 0-8GB
    - + **50%** of visible target memory from 8Gb-64GB
    - + **25%** of visible target memory > 64GB
  - SQL Server 2000
    - SQL Server 2000 4GB upper cap on the plan cache
  - 32-bit? AWE memory is not “visible” to the plan cache (plan cache has to live in “real memory”)

| 2008/2005 SP2 |            |
|---------------|------------|
| Memory        | Plan Cache |
| 4GB           | 3.0GB      |
| 8GB           | 3.5GB      |
| 16GB          | 4.2GB      |
| 32GB          | 5.8GB      |
| 64GB          | 9.0GB      |
| 128GB         | 12.2GB     |
| 256GB         | 30.0GB     |

## Statement Execution

- **Ad hoc queries/batches**
  - Might become auto-parameterized queries
- **Dynamic string execution (DSE)**
  - EXECUTE (@string)
- **Forced statement caching:**
  - sp\_executesql
  - Prepared queries (forced statement caching)
  - Client-side caching from ODBC and OLEDB
    - Exposed via SQLPrepare / SQLExecute and ICommandPrepare
- **Stored procedures (and triggers)**

## Statement Execution Simplified

Optimization = Compilation



*Optimization: this is really where you can have the greatest impact and where the most interesting events can occur...*

1. Parse
2. Standardization/normalization/algebrization  
⇒ Query tree (not T-SQL anymore)
3. Cost-based optimization (statistics are used to come up with optimization plan as well as lock granularity)
4. Compilation
5. Execution: at runtime, if the resources don't exist to support the lock level then escalation occurs to TABLE level (by default)

## Statement Execution: Statistics Events

Optimization = Compilation

 hidden slide  
w/extra details



### Missing Statistics Event

If `auto_create_stats` is enabled then SQL Server (the QO) will WAIT while statistics are created

### Invalidated Statistics Event

- If `auto_update_stats_async` is disabled AND `auto_update_statistics` is enabled then SQL Server will WAIT while statistics are created
- If `auto_update_stats_async` is enabled then SQL Server will optimize based on the invalidated statistics AND kick off an update (which will be used by subsequent users)
- If both methods for updating statistics are disabled then the query will optimize using the invalidated statistic and a warning will be generated (visible in [xml] showplan)

## Statement Execution Simplified

### Some Interesting Notes/Myths

 hidden slide  
w/extra details

#### During standardization:

- Views are “expanded” to their base tables.
- \* is expanded to entire column list (even in an EXISTS clause). This is where the “folklore” of EXISTS (SELECT 1...) comes from. Yes, you can save THAT step but it’s NOT a significant improvement in performance at all.
- The hint “noexpand” is used to force SQL Server to look to the view for indexing/tuning instead of expanding the views. And, the opposite – EXPANDVIEWS essentially allows you to force indexes on the base tables (instead of using the index on the view).

#### During optimization:

- The values of the adhoc statement help define what plan will be chosen... With other methods of execution OTHER things can happen. EG. When a procedure is executed the values used on first execution (when there are no plans already in cache) are “sniffed” to provide the plan for the proc. Subsequent executions will use that plan regardless of their parameter values. This can be good. This can be bad.

## Statement Auto-Parameterization

### Much Ado About Nothing!

- Lots of hype, not really a lot of benefits...
- **Idea:** automatically parameterize search arguments and place statement in cache for future re-use of plan
- **Problem:** queries are only placed in the cache when they're "safe"
  - Needs to be a VERY straightforward statement/plan.
  - Many limitations:
    - FROM clause cannot have more than one table
    - WHERE clause cannot have expressions joined by OR
    - WHERE clause cannot have an IN clause
    - Statement cannot contain a sub-query
    - VERY restrictive (see Appendix A in whitepaper for complete list)
  - Parameters do not change plan choice

## Statement Auto-Parameterization

### Much ado about nothing!

- **How does parameterization work by default: SIMPLE**
  - Most statements are probably NOT deemed safe
- **Database option: parameterization FORCED**
  - Generally, not recommended
  - Many more statements are forced to be cached
    - **PRO:** if you have stable plans from a lot of adhoc clients then this might help to significantly reduce CPU
    - **CON:** If you have some statements that really aren't safe, you could end up executing bad plans... better to do this right from the start and control it yourself with stored procedures (much more difficult!!)
    - **Recommendation:** can investigate plan stability by using query\_hash and query\_plan\_hash (more coming up)
- **If statement is parameterized and safe (or forced), SQL Server places statement in cache for subsequent executions**

## Understanding sp\_executesql

- Usually used to help build statements from applications
- Parameters are typed explicitly
- Forces a plan in cache for the parameterized string – subsequent executions will use this plan
  - **Important note:** this is *not* always a good thing...
- *Almost* like dynamic string execution, but it's not!

## Dynamic String Execution (DSE)

- String is NOT evaluated until runtime (execution)
- Parameters allow virtually any statement to be built “dynamically”
- String can be up to 2GB in size
  - 2000: Had to concatenate multiple variables
  - 2005+: Can declare variable of type (n) varchar(max)
  - sp\_executesql only allows parameters where a typical SQL statement would allow them; however, these two can be combined!
- Can be complex to write, read, perform
- And there's a whole discussion about security...

## Query Plans/Plan Cache Problems

- Plan cache pollution and single-use plans
- Take the following identical query (except for the SARG):
  - SELECT ... FROM member WHERE lastname = 'Tripp'
  - SELECT ... FROM member WHERE lastname = 'Tripps'
  - SELECT ... FROM member WHERE lastname = 'Tripped'
  - SELECT ... FROM member WHERE lastname = 'Tripper'
  - SELECT ... FROM member WHERE lastname = 'Falls'
- Each plan takes 24,385 bytes (or roughly 24K)
- This “query class” is harder to track because each is listed in the cache... but, they will all have the same query\_hash but possibly not 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 then the statement may in fact be safe. However, this is completely data dependent.

## The “Cumulative Effect” of Queries

- SQL Server 2008 adds query\_plan and query\_plan\_hash to sys.dm\_exec\_query\_stats
  - Aggregate by query\_hash to find similar queries
  - Aggregate by query\_hash, query\_plan\_hash to find similar queries along with their plans (possibly more than one per query\_hash which is why the statement wasn't safe)
- SQL templatises the parameters – similar to sp\_get\_query\_template
- See the query on the next slide and the demo scripts...
- Check out BOL: Finding and Tuning Similar Queries by Using Query and Query Plan Hashes
  - <http://bit.ly/QfUmRY>

## The “Cumulative Effect” of Queries

```
SELECT qs2.query_hash AS [Query Hash]
      , SUM(qs2.total_worker_time)/SUM(qs2.execution_count)
        AS [Avg CPU Time]
      , COUNT(*) AS [Number of plans]
      , MIN(qs2.statement_text) AS [Statement Text]
FROM
  (SELECT qs.*,
    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) AS
      statement_text
    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 ORDER BY 2 DESC;
```

## 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 – can consider forced parameterization
    - Reduces I/Os required to access the data the query needs
    - Reducing [often, drastically] I/Os translates into:
      - 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 IOs. Review the showplan and statistics time as well!!)*
- Fewer pages in cache results in better cache efficiency

## Verifying Plans in Cache NOW

- **Use syscacheobjects (in SS2000 and SS2005)**
  - `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 SS2005+**
  - 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

## Plan Cache

- **CACHESTORE\_OBJCP = "Object Plans"**
  - Stored procedures, functions, triggers...
  - Generally, desirable
- **CACHESTORE\_SQLCP = "SQL Plans"**
  - Ad-hoc SQL statements (incl. parameterized ones)
  - Prepared statements
  - OK when highly re-used but often not reused...
- **CACHESTORE\_PHDR = "Bound Trees"**
  - Views, constraints and defaults
  - Irrelevant here, usually OK
- **CACHESTORE\_XPROC = extended procs**

## Verifying State of Plan Cache

```
SELECT objtype
, count(*) AS [Total Plans]
, sum(cast(size_in_bytes AS decimal(12,2)))/1024/1024
AS [Total MBs]
, avg(usecounts) AS [Avg Use Count]
, sum(cast((CASE WHEN usecounts = 1 THEN size_in_bytes ELSE 0 END) AS
decimal(12,2)))/1024/1024
AS [Total MBs - USE Count 1]
, sum(CASE WHEN usecounts = 1 THEN 1 ELSE 0 END)
AS [Total Plans - USE Count 1]
FROM sys.dm_exec_cached_plans
GROUP BY objtype
ORDER BY [Total MBs - USE Count 1] DESC

SELECT cp.*, st.text
FROM sys.dm_exec_cached_plans AS cp
CROSS APPLY sys.dm_exec_sql_text(cp.plan_handle) AS st
ORDER BY objtype
```

## Clearing Plans from Cache

- **DBCC FREEPROCCACHE [ ( { plan\_handle | sql\_handle | pool\_name } ) ]**  
**[WITH NO\_INFOMSGS]**
- **All plans**
  - DBCC FREEPROCCACHE
- **A single plan (using plan\_handle)**
- **A single database's plans**
  - DBCC FLUSHPROCINDB(DBID)
- **Ad hoc plans**
  - In 2005+ use: DBCC FREESYSTEMCACHE('SQL Plans')
  - In 2008+ use: sp\_configure 'optimize for ad hoc workloads', 1

## Reducing Plan Cache Pollution (1)

- **Server setting: Optimize for adhoc workloads**
  - On first execution, only the query\_hash will go into cache. For the prior query this is only 380 bytes (compared to the 24K of the plan)
  - On second execution (if), the plan will be placed in cache
- **Create a single and more consistent plan with covering indexes – might make the plan more stable!**
  - A lot of limitations to SQL Server detecting this as safe (see Appendix A of the “Plan Caching in SQL Server 2008” whitepaper) but if the plans are actually stable...
  - Consider database setting: forced parameterization
    - Not as highly recommended but if you’re finding A LOT of single-use statements that are executed frequently but **with only one query\_plan\_hash** then this might be great!

## Reducing Plan Cache Pollution (2)

### Two primary scenarios to consider

- **Analyze the plan cache for the number of query plans per query hash (as well as the number of executions)**

```
SELECT qs.query_hash
      , COUNT(DISTINCT qs.query_plan_hash) AS [Distinct Plan Count]
      , SUM(qs.EXECUTION_COUNT) AS [Execution Total]
FROM sys.dm_exec_query_stats AS qs
CROSS APPLY sys.dm_exec_sql_text(sql_handle) AS st
CROSS APPLY sys.dm_exec_query_plan(plan_handle) AS qp
WHERE st.text LIKE '%member%'
GROUP BY qs.query_hash
ORDER BY [Execution Total] DESC
```

#### Scenario 1

| query_hash         | # Plans | # Executions |
|--------------------|---------|--------------|
| 0x04BB791B589774AD | 1       | 6456456      |
| 0x1706E9EC3049A95B | 6       | 276543       |
| 0x5BD9FF487079B335 | 1       | 124345       |
| 0x6604520C5200ABC0 | 1       | 78905        |
| 0x77BA5A89C7EBE605 | 1       | 14342        |
| 0xA078B4BC8768A9A6 | 1       | 4567         |
| 0xB81E270A58A79D16 | 1       | 6            |

Mostly stable plans (only 1 plan per hash)

#### Scenario 2

| query_hash         | # Plans | # Executions |
|--------------------|---------|--------------|
| 0x04BB791B589774AD | 34      | 6456456      |
| 0x1706E9EC3049A95B | 1       | 276543       |
| 0x5BD9FF487079B335 | 8       | 124345       |
| 0x6604520C5200ABC0 | 3       | 78905        |
| 0x77BA5A89C7EBE605 | 2       | 14342        |
| 0xA078B4BC8768A9A6 | 9       | 4567         |
| 0xB81E270A58A79D16 | 24      | 6            |

Mostly UNstable plans (multiple plans per hash)

**Scenario 2**  
*Generally,  
more likely...*

## Reducing Plan Cache Pollution (3)

- **Scenario 2: Default parameterization mode: SIMPLE**

- Use “templatized” plan guides to take the few statements that are safe and make them forced (reduced compilation/CPU)

```
EXEC sp_create_plan_guide
    Name_of_plan_guide
    templatized_version_of_safe_query,
    N'TEMPLATE',
    NULL,
    @Parameters,
    N'OPTION(PARAMETERIZATION FORCED)';
```

- **Scenario 1: Consider changing parameterization to FORCED**

- Use “templatized” plan guides to take the few statements that are NOT safe and make SIMPLE (recompiled)

```
EXEC sp_create_plan_guide
    Name_of_plan_guide
    templatized_version_of_unsafe_query,
    N'TEMPLATE',
    NULL,
    @Parameters,
    N'OPTION(PARAMETERIZATION SIMPLE)';
```



23

© SQLskills. All rights reserved.  
<http://www.SQLskills.com>

## But is this Really the Best Way?

### A lot of work and a lot of management/overhead...

- **Ad-hoc/generated statements are not ideal**

- Can create plan cache bloat
- Requires compilation/CPU due to lack of parameterization (most statements are **not** deemed safe)
- Different parameters **warrant** different plans – for the best performance
- Only guaranteed reuse are exact textual matches

- **For better plan re-use consider:**

- Forced statement caching through sp\_executesql
- Stored procedures

- **For better plan re-use, flexibility and centralized logic/control use:**

- Stored procedures

*With statement-level recompilation techniques based on server-side knowledge, testing, and the characteristics of the data*



24

© SQLskills. All rights reserved.  
<http://www.SQLskills.com>

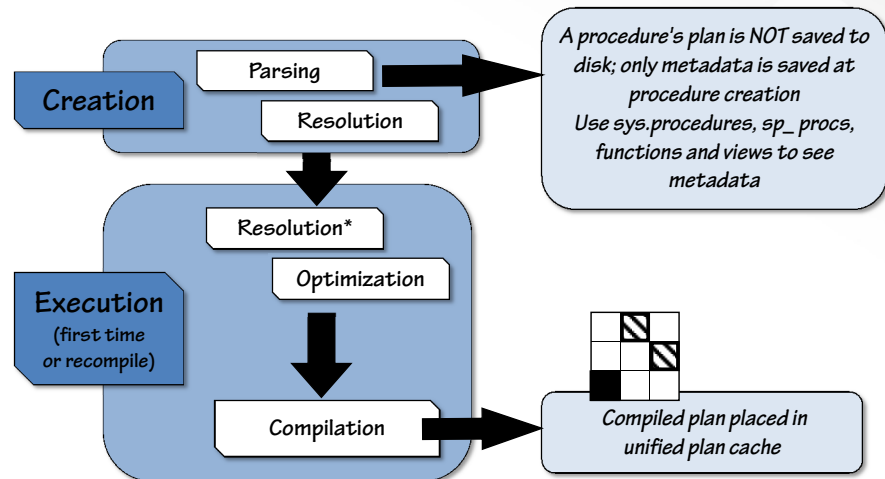
## Overview

- Part I: Understanding/Analyzing Plan Cache
- **Part II: What's in the Plan Cache?**
  - A few demos of things you can do WITH the cache...
- Part III: Understanding/Optimizing Stored Procedures
- Part IV: A Cautionary Tale about Scalar Functions
- Part V: Index Analysis

## Overview

- Part I: Understanding/Analyzing Plan Cache
- Part II: What's in the Plan Cache?
- **Part III: Understanding/Optimizing Stored Procedures**
  - Stored procedure creation
  - Stored procedure coding best practices for performance
  - Stored procedure execution
  - Stored procedure recompilation/optimization
- Part IV: A Cautionary Tale about Scalar Functions
- Part V: Index Analysis

## Processing Stored Procedures



## Stored Procedure Optimization

- Plan is generated when no plan already exists in cache
- Plans are never saved on disk and can "fall out" of cache
  - Forced out through:
    - Server restart
    - DBCC FREEPROCCACHE
    - DBCC FLUSHPROCINDB
    - sp\_recompile
  - Aged out through non-use
  - Schema of base object changes
  - Statistics of base objects change
    - See next slide for details specific to version
- When a plan exists in cache, all subsequent executions use that plan
- Plans are not regenerated for subsequent executions (even when statistics or indexes are added – use sp\_recompile)

## Plan Invalidation

### A Painful History

- In SQL Server 2012, statistics must get updated in order to cause plan invalidation; update statistics does nothing unless you have changed data
- In prior versions (SQL Server 2008R2, 2008, 2005)
  - If database option: `auto_update_stats` is ON
    - Updating statistics always ran and always caused plan invalidation
  - If database option: `auto_update_stats` is OFF
    - Updating statistics always ran but did NOT cause plan invalidation without the database option
- **Recommendation**
  - In earlier versions, use `sp_recompile` after updating stats against a table
  - In SQL 2012, use `sp_recompile` after adding indexes or adding/updating statistics
- **For more information, check out these blog posts:**
  - *What caused that plan to go horribly wrong – should you update statistics?*  
<http://www.sqlskills.com/blogs/kimberly/what-caused-that-plan-to-go-horribly-wrong-should-you-update-statistics/>
  - *Statistics and Recompilations:* <http://erinstellato.com/2012/01/statistics-recompilations/> and Part II: <http://erinstellato.com/2012/02/statistics-recompilations-part-ii/>



29

© SQLskills. All rights reserved.  
<http://www.SQLskills.com>

## Procedure Caching

### Isn't That the Point?

- **Reusing plans can be good**
  - When different parameters don't change the optimal plan, then saving and reusing is excellent!
  - SQL Server saves time in compilation
- **Reusing plans can be VERY bad**
  - When different parameters wildly change the size of the result set and the optimal plans vary, then reusing the plan can be horribly bad
    - Don't worry, I'll show you how to see this...
    - Don't worry, I'll show you the plethora of options to control/fix this!
  - If indexes are added to base tables, existing plans may not leverage them
    - Don't worry, there's a simple solution here:
      - `EXEC sp_recompile <tablename>`



30

© SQLskills. All rights reserved.  
<http://www.SQLskills.com>

## Recompilation Issues

**RECOMPILATION = OPTIMIZATION**

**OPTIMIZATION = RECOMPILATION**

- When do you want to recompile?
- What options do you have for recompilation – and at what granularity?
- How do you know you need to recompile?
- Do you want to recompile the entire procedure or only part of it?
- Can you test it?

## When to Recompile?

- When the plan for a given statement within a procedure is not consistent in execution plan, due to parameter changes
- Cost of recompilation might be significantly less than the execution cost of a bad plan!
- Why?
  - MUCH faster execution with a better plan
  - Some plans just don't work for a wide variety of your execution cases, in fact, some plans should NEVER be saved
- Do you want to do this for every procedure?
  - **No, but start with the highest priority/expensive procedures that aren't performing well first – and test!!**

## Options for Recompilation

- **CREATE ... WITH RECOMPILE**
- **EXECUTE ... WITH RECOMPILE**
- **sp\_recompile objname**
- **Statement-level recompilation**
  - The 2000+ way (still has benefits)
    - Dynamic string execution (statement isn't stored with proc plan)
    - Modularized code (breaks the statement/block into its own proc)
  - The new way
    - 2005+: `OPTION (RECOMPILE)`
    - 2005+: `OPTION (OPTIMIZE FOR (@variable_name = constant, ...))`
    - 2008+: `OPTION (OPTIMIZE FOR UNKNOWN)`

## EXECUTE WITH RECOMPILE

- **Excellent for testing**
- **Verify plans for a variety of test cases**

```
EXEC dbo.GetMemberInfo
'Tripp' WITH RECOMPILE
or 'T%' WITH RECOMPILE
or '%T%' WITH RECOMPILE
```
- **Do the execution plans match?**
  - Yes: create the procedure normally
  - No: determine what should be recompiled

## CREATE ... WITH RECOMPILE

- When procedure returns widely-varying results
- When the plan is not consistent
- For SMALL procedures
- For backward compatibility
- Why?
  - 2000: only complete procedure recompiles
  - 2005: statement-level recompilation
- Always target the smallest amount possible to recompile!

## Statement-Level Recompilation

- The old way: modularizing your code
  - Doesn't hurt!
  - Better for statement blocks/code reuse
- In 2005 and 2008+: "inline" recompilation for statements
  - OPTION (RECOMPILE)
    - Excellent when parameters cause the execution plan to widely vary
    - Bad because EVERY execution will recompile – but only for the statements
  - OPTION (OPTIMIZE FOR (@variable = literal, ...))
    - Excellent when large majority of executions generate the same optimization time
    - You don't care that the minority may run slower with a less than optimal plan?
  - 2008+ only: OPTION (OPTIMIZE FOR UNKNOWN)
    - Use the "all density" (average) instead of the histogram

## Modularization

- **Be careful of block statements/conditional logic**
  - SQL Server optimizes the process of optimization and this may lead to an less-than-optimal plan (no, really!)
- **Breaking the stored procedure into smaller chunks**
  - Can handle the block of statements on a more granular basis
  - Subprocedures can be optimized/compiled
    - `CREATE subprocedure WITH RECOMPILE`
    - `EXECUTE subprocedure WITH RECOMPILE`

## 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
  - Define the data type of the variable in the string. Use: `column = convert(datatype, @variable)` in actual string (to reduce implicit conversions and plan cache bloat)
  - Be careful using `sp_executesql`

## Ad Hoc Statements and Statement Auto-Parameterization

- **EXEC (@str) = ad hoc statements, these can:**
  - Be cached when there are no parameters: subsequent executions must be an exact textual match (case-sensitive regardless of db setting and spaces)
  - Be parameterized and deemed safe
    - Subsequent executions will use the plan in cache
  - Be parameterized and deemed unsafe (MOST LIKELY)
- **sp\_executesql = forced statement caching**
  - The first execution will be “sniffed” and the plan will be placed in cache for subsequent executions, regardless of whether or not the plan is good/consistent for all values
  - Great when the plan is consistent!
  - Can be horrible when it’s not!!

## 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!
- But, can you have too many recompiles?
  - Yes: however, it’s not as bad as 2000 because only the statement is recompiled (as of 2005+), not the entire procedure
  - Yes: however, sticking to these best practices can help to **DRASTICALLY** minimize that!

## Overview

- Part I: Understanding/Analyzing Plan Cache
- Part II: What's in the Plan Cache?
- Part III: Understanding/Optimizing Stored Procedures
- **Part IV: A Cautionary Tale about Scalar Functions**
  - Table-valued vs. Scalar Functions
  - Functions vs. Stored Procedures
  - When Scalar Functions go WRONG!
- Part V: Index Analysis

## Table-valued vs. Scalar Functions

- **Table-valued functions**
  - Single statement: like a view with parameters
  - Multi statement: similar in appearance to stored procedure, but not as optimizable
- **Scalar functions**
  - Return single value
  - Executed once-per-row
  - Database access is optional (e.g. formulas)
  - If deterministic and precise can be used in
    - Indexed views
    - Persisted computed columns (SQL 2005+)

## Functions vs. Stored Procedures

- Scalar functions force row operations
- Stored procedures can perform set operations
- Table functions cannot use INSERT EXEC to populate the table variable
- Table function's table variables can only have "key" indexes created
- Tables created in stored procedures can have any type of index created (i.e. non-unique/comp)

## "Simple" Expressions...

- Imagine a simple scalar which takes a member\_no and looks up their corresponding name, concatenating their first and last name to return a nice "Firstname Lastname" string:

```
CREATE FUNCTION dbo.MemberName
(@MemberNo int)
RETURNS varchar(31)
AS BEGIN
RETURN
    (SELECT m.FirstName + ' ' + m.LastName
     FROM dbo.Member AS m
     WHERE m.member_no = @MemberNo)
END
```

## First Case

- **Query the table/put the expression in the SELECT LIST**

```
SELECT m.member_no  
      , m.FirstName + ' ' + m.LastName  
FROM dbo.member AS m
```

- Pro: faster
- Con: more “complex” query

- **Query the table/put the function in the SELECT LIST**

```
SELECT m.member_no  
      , dbo.MemberName(m.member_no)  
FROM dbo.member AS m
```

- Pro: simplified query
- Con: SLOW

## More Interesting Case (1)

- **Query the table and put the expression in the SELECT LIST – WITH A JOIN**

```
SELECT c.member_no  
      , m.FirstName + ' ' + m.LastName  
      , sum(c.charge_amt)  
FROM dbo.charge AS c  
      JOIN dbo.member AS m  
      ON c.member_no = m.member_no  
GROUP BY c.member_no  
      , m.FirstName + ' ' + m.LastName
```

- Pro: faster
- Con: more “complex” query

## More Interesting Case (2)

- Query the table and put the function in the SELECT LIST – NO JOIN

```
SELECT c.member_no
      , dbo.MemberName(c.member_no)
      , sum(c.charge_amt)
FROM   dbo.charge AS c
GROUP BY c.member_no
      , dbo.MemberName(c.member_no)
```

- Pro: simplified query
- Con: **PAINFULLY SLOW**

## Showplan and Functions

- TWO concerns

- Estimates (and therefore the percentages in showplan)
  - They don't correctly cost-estimate the function so any plan that includes one is going to be estimated lower than it likely is (and possibly WAY lower when there's data access involved)
- Executing with the "display actual showplan" on
  - Since "actual" requires execution in a different mode – the actual execution is even more expensive... so, not only is the plan wrong but if you ask for the plan, it takes even longer to execute
    - 1 min, 8 secs (without showplan)
    - 18 mins, 25 secs (with showplan)

- To see more information, check out the statement executions in cache

- You'll have one statement for EVERY execution (you'll be able to catch these by looking at the cumulative effect of your queries – using query\_hash)

## Overview

- Part I: Understanding/Analyzing Plan Cache
- Part II: What's in the Plan Cache?
- Part III: Understanding/Optimizing Stored Procedures
- Part IV: A Cautionary Tale about Scalar Functions
- **Part V: Index Analysis**
  - Index Cleanup
  - Index Health
  - Missing indexes

## Index Analysis

- **Index cleanup**
  - Get rid of the dead weight
  - Remove duplicate indexes
  - Remove unused indexes
- **Index health**
  - Determine health of existing (and useful) indexes
- **Missing indexes**
  - Slowly and iteratively – add missing indexes
  - Database Tuning Advisor
  - Performance Data Collection

## (1) Remove Unused Indexes

### DMVs Don't Tell You Everything...

- Removing redundant/duplicate indexes: *SQL Server allows you to create as many useless indexes as you like...*
- Duplicate indexes can still show as used
- **MUST** review your indexes manually
  - Don't forget INCLUDED columns (in 2005) or filtered indexes (in 2008)
  - sp\_helpindex doesn't show these columns
  - Use my updated version of sp\_helpindex (blog category: sp\_helpindex rewrites) to get better information and determine if one index really is redundant/duplicate
  - Kimberly's blog post: [Removing duplicate indexes](http://bit.ly/rusl9U) (<http://bit.ly/rusl9U>)
- Don't rely on sys.dm\_db\_index\_usage\_stats alone as it lies

## (1) Remove Unused Indexes

### DMVs Don't Tell You Everything...

- In addition to completely duplicate indexes, must prune out the redundant indexes...
- Pruning existing indexes is not quite as simple as removing all left-based subsets:
  - It's true that a query using an index on LastName alone COULD use an index on LastName, Firstname OR an index on LastName, Firstname, MiddleInitial but, always be careful of how much larger the indexes are and the type of queries using them
  - Should you drop indexes that are left-based subsets of others?
    - Typically yes BUT, consider the width of the additional columns
    - If the additional column(s) are relatively wide and only needed for a couple of queries whereas the narrower version is used by a lot of queries, you might want both!

## (1) Remove Unused Indexes

### What Do the DMVs Tell You?

- Verify index usage with `sys.dm_db_index_usage_stats`
- Tracks the following and the date/time of their last occurrence:
  - Seeks (a singleton lookup or range scan)
  - Scans (something like a 'select \*')
  - Lookups (a bookmark lookup)
  - Updates (an insert, update, or delete)
- The cache is flushed at shutdown as well as when
  - An object's index usage stats are cleared when the object is REBUILD (not reorged) in SQL Server 2012 (check EACH SP)
    - Joe Sack blog post on this: <http://bit.ly/JdunGW>
  - Entries for all indexes in a database are removed when the database is closed (via AUTOCLOSE (if enabled)), taken offline, or detached
  - There's no way to manually flush the cache
- Persist this data and analyze over business cycle

## (1) Dropping an Index

### What Could Go Wrong?

- Queries that use index hints will ERROR if the index no longer exists
  - This is the reason for why SQL Server allows duplicate indexes to be created in general...
- Plans guide might no longer work
  - They're just invalidated, a new plan will be used
- Plans could change
  - This could be good... or bad?!

## (2) Verify Health of Existing (and Useful) Indexes...

- Fragmentation can mean a lot of things (completely overloaded term) and not an entirely simple thing to address...why? It depends on many factors...
  - Table size and usage patterns
  - Impact to availability – can you use an online operation?
    - ALTER INDEX...REBUILD...WITH (ONLINE = ON)
      - Not if the index has a LOB column in it
      - Not if you try to rebuild only a single partition
    - ALTER INDEX...REORGANIZE (always online)
  - Reorganizing always uses 'full' logging but the amount of log information generated will depend on how much fragmentation exists
  - There are trade-offs between rebuilding and reorganizing in terms of log space, disk space, run-time, impact to tempdb and even benefit...
  - Review the [Microsoft SQL Server 2000 Index Defragmentation Best Practices](#) whitepaper as concepts and techniques still apply
  - Review the [Online Indexing Operations in SQL Server 2005](#) whitepaper for details about online index operations

## (2) Verify Health of Existing (and Useful) Indexes...

- Verify structural details (levels/fragments/density) from `sys.dm_db_index_physical_stats`
  - Can run in one of three modes:
    - LIMITED (default): the logical (left to right) order of the pages (as defined by the index keys) is not the same as the physical order in which the pages are allocated
    - SAMPLED: run in detailed mode for tables < 10000 pages but for tables with >= 10000 pages, "samples" every 100th page to get a picture of the index fragmentation (faster but less accurate for some stats)
    - DETAILED: slowest, but shows ALL structural and fragmentation details
  - Tracks the following:
    - avg\_fragmentation\_in\_percent: overall health of the index...this number should be very low and it should not change rapidly
    - avg\_page\_space\_used\_in\_percent: overall health of the pages...this should be relatively high but might have some freespace (fillfactor) that's specifically been added in an attempt to reduce overall fragmentation
- Check out our blog post categories on Indexes:  
Kimberly: <http://www.sqlskills.com/blogs/Kimberly/category/Indexes.aspx>  
Paul: <http://www.sqlskills.com/blogs/paul/category/Indexes-From-Every-Angle.aspx>

## (2) Verify Health of Existing (and Useful) Indexes...

- **Verify operational details (stats/waits/overflow) from `sys.dm_db_index_operational_stats`**
  - Tracks the following:
    - `Leaf_X_count` (essentially rows inserted, deleted or updated – in the leaf level of the index)
    - `nonleaf_X_count` (this indicates that pages were added/removed in the b-tree...this might indicate heavy fragmentation but you **MUST** review the index physical stats to be sure)
    - Significant overflow activity could indicate poor IO patterns
    - `row_lock_wait_in_ms` and `page_lock_wait_in_ms` show the TOTAL amount of blocking on these structures – this could indicate poor index choices and/or a need to change reader/writer isolation options
    - `page_io_latch_wait_count` and `page_io_latch_wait_in_ms` show the total physical I/Os that were necessary to access the data
- **The cache is flushed when the object falls out of metadata cache... this cannot be predicted (per se) but “active” objects will be available in this DMV when queried**
- **Consider using a custom data collection set in Data Collection to persist snapshots of this (and possibly other) DMVs for better trend analysis**

## (3) Are You Missing Any Indexes?

- **Have you tried other options?**
- **Ask DTA what it thinks?**
- **Ask SQL Server what it thinks?**
  - SQL Server 2005
    - DMVStats
    - Performance Dashboard Reports (SP2)
    - RML Utilities from PSS
  - Both SQL Server 2005+
    - DMV queries
    - Other resources/blogs/sites...
  - SQL Server 2008+
    - [Performance] Data Collector

### (3) Are You Missing Any Indexes?

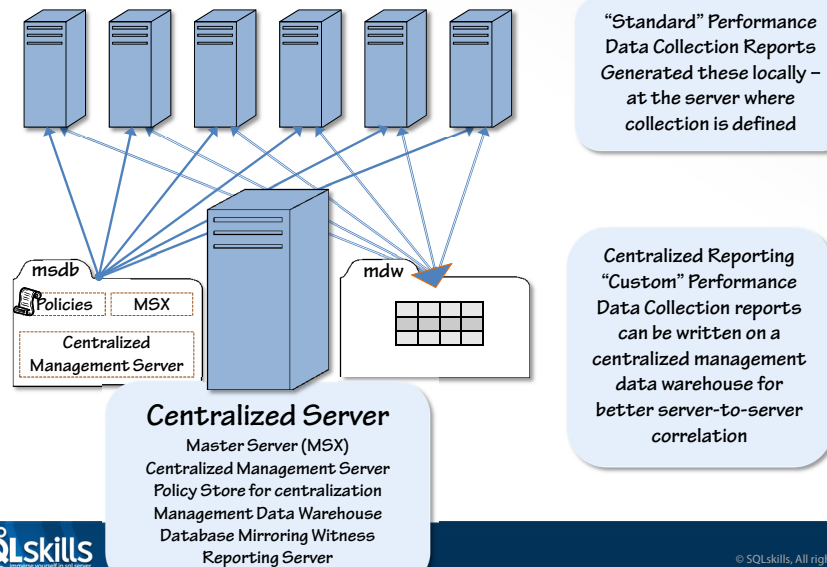
#### Ask SQL Server What it Thinks...

- **sys.dm\_db\_missing\_index\_group\_stats** (probably the most detailed):
  - user\_seeks, user\_scans
  - last\_user\_seek and last\_user\_scan are both datetime
  - avg\_total\_user\_cost: higher costs give relative numbers to determine which are more “costly” to the system
  - avg\_user\_impact: the improvement (in terms of percent that the user should see from the index addition)
- **sys.dm\_db\_missing\_index\_groups**
  - Many to many relationship table to tie together the index details and the usage stats (index group stats is effectively the index usage stats for these needed indexes)
- **sys.dm\_db\_missing\_index\_details**
  - Details the table, key columns and included columns that you should consider for these indexes...
- **NOTE: The important part isn't the query, it's what you do with the results (evaluation, testing, consolidation)**

### (3) Asking the Tools

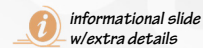
- **USE the tools!**
  - STATISTICS IO
  - Showplan/missing index DMVs
  - Database [Engine] Tuning Advisor
- **BEWARE of the limitations of the tools!**
  - Missing index DMVs (and therefore showplan) only tune the plan that was executed; they do not “hypothesize” about alternatives (like DTA does)
  - All of the index recommendation from tools tend to go for “the best” choice rather than good enough choices
  - NONE of the tools do index consolidation...
- **Resources:**
  - Search “Bart Duncan Missing”
  - Find Missing Indexes on SQLServerPedia
    - [http://sqlserverpedia.com/wiki/Find\\_Missing\\_Indexes](http://sqlserverpedia.com/wiki/Find_Missing_Indexes)
  - A bit of searching as lots of good stuff out there (still coming!)

### (3) Centralized Administration Data Collection and Reporting



### Performance Dashboard Reports on 2008/R2

#### A Quick Side Note/FYI



- SQL Server 2005 Performance Dashboard Reports (download from Microsoft Download Center)
- Install for 2005, copy the PerformanceDashboard directory to 100\Tools
  - 2005 path: C:\Program Files (x86)\Microsoft SQL Server\90\Tools\PerformanceDashboard
- Change the setup.sql
  - cpu\_ticks\_in\_ms -> ms\_ticks
  - 64bit datediff problem: <http://theskythelimit.blogspot.com/2008/03/fix-difference-of-two-datetime-columns.html> (<http://bit.ly/RqZerE>)
- Use the modified script in your script directories

## Performance Dashboard Reports on 2012

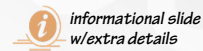
### A Quick Side Note/FYI



- SQL Server 2012 Performance Dashboard Reports have been updated (download from Microsoft Download Center)
  - <http://www.microsoft.com/en-us/download/details.aspx?id=29063>

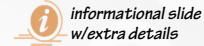
## Performance Dashboard Reports

### A GENERAL Side Note/FYI



- The Good (quick/easy)
  - Looks at current, in-memory state
  - Good for overall server, generally bad queries, digging in and seeing some info but a lot of good info is hard to get
  - Easy to get at and easy to use if you want to get some insight into what's happening **at a particular moment**
- The Bad
  - No historical information (PDC/PA – info on next slide)
  - Only good for “right now”
  - Doesn't give you access to the graphical plan directly: must export (only options are: Excel or pdf – *are you kidding??*) OR manually type in the sql/plan\_handle and that's **iff** it's still in cache

## Performance Data Collection Resources/Why it's Not Covered in Class



- **For Performance Data Collection**
  - Microsoft eLearning Resources - Clinic 10259 (SQL 2008 DBIS)
    - [http://www.sqlskills.com/BLOGS/KIMBERLY/post/Microsoft-eLearning-Resources-Clinic-10259-\(SQL-2008-DBIS\).aspx](http://www.sqlskills.com/BLOGS/KIMBERLY/post/Microsoft-eLearning-Resources-Clinic-10259-(SQL-2008-DBIS).aspx) (<http://bit.ly/amRwEm>)
  - Performance Data Collection LAB
- **NOTE: There are many inefficiencies in PDC and no enhancements in the latest version (2012); that's why it's not a topic covered directly in class. Check out this great article for a comparison to SQL Sentry's Performance Advisor (similar but MUCH more extensive and efficient):**
  - Monitoring Your Servers : Data Collection vs. Performance Advisor, Aaron Bertrand
    - <http://www.sqlservercentral.com/articles/Product+Reviews/66278/>
  - Remember, you can download PE for free and you can also get a PA evaluation license for a 45 day trial as an Immersion Event attendee!

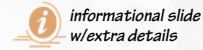
## Summary: Indexing for Performance

- **Extremely challenging**
  - Users lie ☺
  - Workload specific
    - Data modifications are impacted by indexes (indexes add overhead to INSERTs/UPDATEs/DELETEs)
    - The type and frequency of the queries needs to be considered
      - This can change over time
      - This can change over the course of the business cycle
  - Need to have an understanding of how SQL Server works in order to create the "RIGHT" indexes, you CANNOT just guess!
- **To do a good job at tuning you must:**
  - Know your data
  - Know your workload
  - Know how SQL Server works!

## Indexing Strategies at Design

This is For Your Developers (Yes, Talk to Them 😊)

- First and foremost: choose a GOOD clustering key
- Create your primary keys and unique keys
- Create your foreign keys
  - Manually index your foreign keys with nonclustered indexes
- Create any nonclustered indexes needed to help with highly selective queries (lookups are OK for highly selective queries)
- **STOP** – this is your “design” base
- Add indexes slowly and iteratively over time while learning and understanding your workload as well as query priorities and always re-evaluate if/when things change!



## Review

- Part I: Understanding/analyzing plan cache
- Part II: What's in the plan cache?
- Part III: Understanding/optimizing stored procedures
- Part IV: A cautionary tale about scalar functions
- Part V: Index analysis

## Plan Cache Pollution Resources

- **Blog posts:**
  - [Plan cache and optimizing for adhoc workloads](#)
  - [Plan cache, adhoc workloads and clearing the single-use plan cache bloat](#)
  - [Clearing the cache - are there other options?](#)
  - [Statement execution and why you should use stored procedures](#)
  - Category: Optimizing Procedural Code
    - <http://www.sqlskills.com/BLOGS/KIMBERLY/category/Optimizing-Procedural-Code.aspx>
- **MSDN Article: How Data Access Code Affects Database Performance, Bob Beauchemin**
  - <http://msdn.microsoft.com/en-us/magazine/ee236412.aspx>



69

© SQLskills, All rights reserved.  
<http://www.SQLskills.com>

## What's in the Plan Cache Resources

- Demo scripts have links in them!
- **\*LOTS\*** of great resources (and they keep coming) from variety of sources online!
- **Finding Implicit Conversions in the Plan Cache**
  - [http://sqlblog.com/blogs/jonathan\\_kehayias/archive/2010/01/08/finding-implicit-column-conversions-in-the-plan-cache.aspx](http://sqlblog.com/blogs/jonathan_kehayias/archive/2010/01/08/finding-implicit-column-conversions-in-the-plan-cache.aspx)
  - Related post: Tibor Karaszi [Match Those Types](#)



70

© SQLskills, All rights reserved.  
<http://www.SQLskills.com>

## Stored Procedure Resources

- Plan Caching in SQL Server 2008
  - <http://msdn.microsoft.com/en-us/library/ee343986.aspx>
- Batch Compilation, Recompilation, and Plan Caching Issues in SQL Server 2005
  - <http://www.microsoft.com/technet/prodtechnol/sql/2005/recomp.msp>
- PSS SQL Server Engine Blog
  - <http://blogs.msdn.com/psssql/default.aspx>
- KB 243586: Troubleshooting Stored Procedure Recompilation
- KB 308737: How to identify the cause of recompilation in an SP:Recompile event (SQL Server 2000 SP2+ has 6 subclass values, SQL Server 2005 has 11 subclass values and SQL Server 2008 RTM has 14 subclass values)
- KB 263889: Description of SQL Server blocking caused by compile locks



71

© SQLskills, All rights reserved.  
<http://www.SQLskills.com>

# Questions?

