

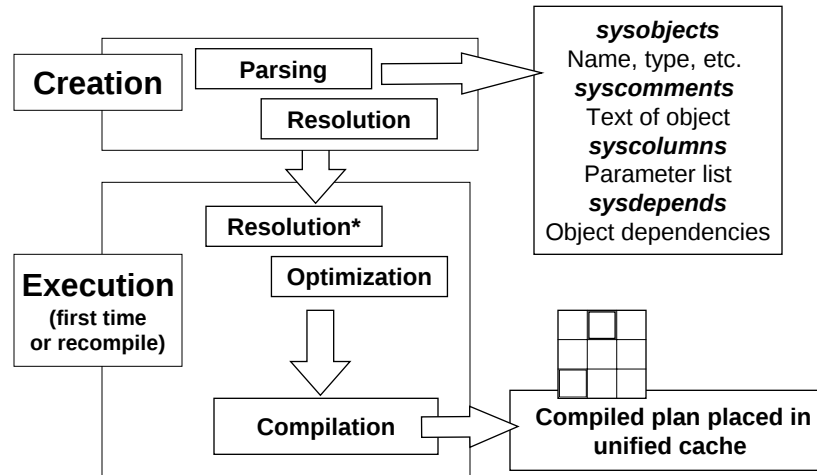
Module 4: Optimizing Procedural Code

Designing for Performance
Kimberly L. Tripp

Overview

- **Initial Processing - Review**
 - Resolution
 - Compilation/Optimization
 - Execution/Recompilation
- **Recompilation Issues**
 - When do you want to Recompile?
 - Options for Recompilation?
 - What to Recompile?
- **User-defined Functions**
 - Functions v. Stored Procedures
 - Scalar functions for simple expressions
- Extra Info: **Stored Procedure Best Practices**

Processing Stored Procedures



Resolution

- When a stored procedure is created all objects referenced are resolved (checked to see whether or not they exist).
- The create will succeed even if the objects do not exist
 - Procedures called that do not exist generate error
Cannot add rows to sysdepends for the current stored procedure because it depends on the missing object 'missingobjectname'. The stored procedure will still be created.
Benefit: Recursion is allowed!
 - Tables, Views, Functions called that do not exist - do NOT generate error (unless in 6.5 compatibility mode)
- Verify dependencies with sp_depends before dropping an object

Compilation/Optimization

- Based on parameters supplied
- Future executions will reuse the plan
- Complete optimization of all code passed (*more on this coming up...modular code!*)
- Poor coding practices can cause excessive locking/blocking
- Excessive recompilations can cause poor performance

Execution/Recompilation

- Upon Execution if a plan is not already in cache then a new plan is compiled and placed into cache
- What can cause a plan to become invalidated and/or fall out of cache:
 - Server restart
 - Plan is aged out due to low use
 - DBCC FREEPROCCACHE
(sometime desired to force it)
- Base Data within the tables - changes:
 - Same algorithm as AutoStats, see Q195565 INF: How SQL Server 7.0 and SQL Server 2000 Autostats Work

Recompilation Issues

RECOMPILE = OPTIMIZATION

OPTIMIZATION = RECOMPILE

- When do you want to recompile?
- What options do you have
Recompilation?
- 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?

Options for Recompilation

- **CREATE ... WITH RECOMPILE**
 - When procedure returns widely varying results
 - When the plan is not consistent
- **EXECUTE ... WITH RECOMPILE**
 - For testing and to determine if CREATE WITH RECOMPILE is necessary
- **sp_recompile *objname***
 - Forces all plans with regard to that object to be invalidated (note: this does not force recompilation on views even though a view name is supported)
- **Statement Recompilation**
Dynamic String Execution or Modularized Code

When to recompile?

- When the plan for a given statement within a procedure is not consistent in execution plan–due to parameter and/or data changes
- Cost of recompilation might be significantly less than the execution cost of a bad plan!

Why?

- Faster Execution with a better plan
- Saving plans for reuse is NOT always beneficial
- Some plans should NEVER be saved

What Should be Recompiled?

- Whole Procedure
 - CREATE with RECOMPILE
 - Procedure is recompiled for each execution
 - EXECUTE with RECOMPILE
 - Procedure is recompiled for that execution

NOTE: Consider forcing recompilation through another technique – you should not expect users will know when/why to use EXECUTE ... WITH RECOMPILE once in production!
- Statement(s) Recompilation
 - If limited number of statements cause recompile
 - Dynamic String Execution
 - Modular Code

How do you know?

You Test!

- Test optimization plans consistency using **EXECUTE WITH RECOMPILE**
- Choose what needs to be recompiled
 - Whole Procedure
 - Portions of the procedure
- Test final performance using the chosen strategy
 - Procedure Recompilation (CREATE with RECOMPILE)
 - Statement Recompilation (Dynamic String Execution)
 - Modularized Code (Sub procedures created with or without WITH RECOMPILE)

EXECUTE WITH RECOMPILE

- Excellent for Testing
- Verify plans for a variety of test cases
 - EXEC dbo.GetMemberInfo 'Tripp' WITH RECOMPILE
 - EXEC dbo.GetMemberInfo 'T%' WITH RECOMPILE
 - EXEC dbo.GetMemberInfo '%T%' WITH RECOMPILE
- Do the execution plans match?
- Are they consistent?
- Yes ⇒ then create the procedure normally
- No ⇒ Determine what should be recompiled

CREATE ... WITH RECOMPILE

- Use when the procedure returns drastically different results based on input parameters.
- May not be the only – or even the best option...
- How do you know?

```
CREATE PROCEDURE dbo.GetMemberInfo
( @LastName    varchar(30) )
    WITH RECOMPILE
AS
SELECT * FROM dbo.Member AS m
    WHERE m.LastName LIKE @LastName
go
EXEC GetMemberInfo 'Tripp'      -- if plan already exists uses it
                                (s/b index+bookmark)
EXEC GetMemberInfo 'T%'        -- if plan already exists uses it
                                (s/b a table scan)
EXEC GetMemberInfo '%T%'       -- if plan already exists uses it
                                (definitely s/b table scan)
```

Statement Recompilation

- What if only a small number of statements need to be recompiled?
- The SQL Statement is not likely safe (i.e. it will not be saved and parameterized)
- Dynamic String Execution!
 - Amazingly Flexible
 - Permission Requirements
 - Potentially Dangerous
 - Advanced Examples
 - Complex large strings
 - Changing database context
 - Output parameters

sp_recompile

- Can be used to periodically and directly force recompilation of a procedure (or trigger)
- Can be used on tables and views to indirectly force the recompilation of all procedures and triggers that reference the specified table or view
- Does not actually recompile the procedures ⇒ Instead it invalidates plans for next execution
- SQL Server invalidates plans as data changes
- Never really negative – especially if you run it at night as part of batch processing after index rebuilds or statistics updates with FULLSCAN

Modular Code

The Better Solution!

IF (expression operator expression)

SQL Statement Block1

ELSE

SQL Statement Block2

Solution?

Do not use a lot of conditional SQL Statement Blocks
Call separate stored procedures instead!

Scenario 1 – upon first execution...

Parameters are passed such that the **ELSE** condition executes – **BOTH** Block1 and Block2 are optimized with the input parameters

Scenario 2 – upon first execution...

Parameters are passed such that the **IF** condition executes – **ONLY** Block1 is optimized. Block2 will be optimized when a parameter which forces the **ELSE** condition is passed.

See ModularProcedures.sql

User-defined Functions

- **Scalar Functions**
 - Similar to a built-in function
 - Returns a single data value built by a series of statements
- **In-line Table-valued Functions**
 - Similar to a view with parameters (Parameterized View)
 - Returns a table as the result of single SELECT statement
 - Look like a view, act like a view – with parameters
- **Multi-Statement Table-valued Functions**
 - Content like a stored procedure
 - Referenced like a view
 - Look like a procedure, act like a view!

Functions v. Stored Procedures

- Functions recompiled each execution
- Stored Procedure plans *can* be controlled in terms of compilation and plan reuse
- 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 create
- Tables created in Stored Procedures can have any type of index created (i.e. non-unique/comp)

See Brian Moran's *Functions* article, May 2003 in SQL Server Magazine (www.sqlmag.com)

Scalar functions for 'simple' expressions...

- Simple Scalar
- One input, one output
- Not just formatting though – it's a lookup.
- The function takes member_no and returns the corresponding name by – concatenating first and last names to return a formatted “Firstname Lastname” string

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

First Case

- Query the table and 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 and 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 complex Case

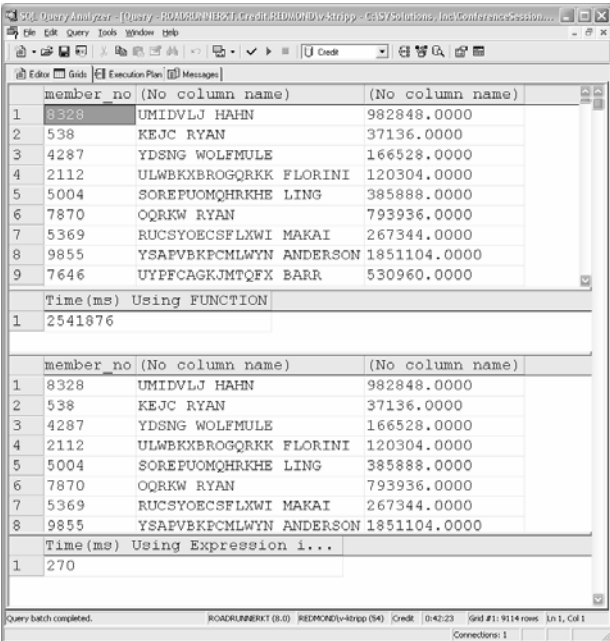
- Query the table and put the expression in the SELECT LIST – WITH A JOIN

```
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
- 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)
```

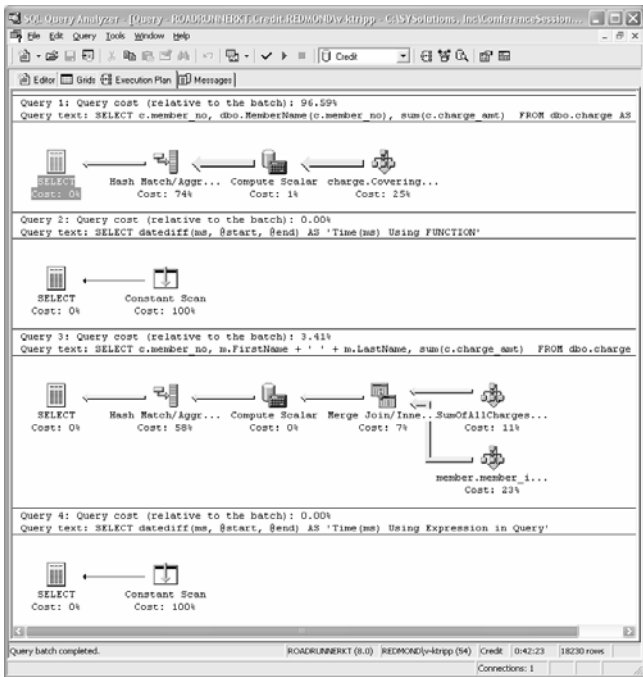
 - Pro – simplified query
 - Con – PAINFULLY SLOW



Didn't really want to have to run this LIVE... ☹

Function query first – more than 42 minutes

Second Query is the join (benefiting from an indexed view)



Query 1: Query cost (relative to the batch): 96.59%
Query text: SELECT c.member_no, dbo.MemberName(c.member_no), sum(c.charge_amt) FROM dbo.charge AS

Query 2: Query cost (relative to the batch): 0.00%
Query text: SELECT datediff(ms, @start, @end) AS 'Time(ms) Using FUNCTION'

Query 3: Query cost (relative to the batch): 3.41%
Query text: SELECT c.member_no, m.FirstName + ' ' + m.LastName, sum(c.charge_amt) FROM dbo.charge

Query 4: Query cost (relative to the batch): 0.00%
Query text: SELECT datediff(ms, @start, @end) AS 'Time(ms) Using Expression in Query'

And the query plans... What's wrong with this?

Is it even *really* 96.59% v. 3.41%

Review

- Initial Processing - Review
 - Resolution
 - Compilation/Optimization
 - Execution/Recompilation
- Recompilation Issues
 - When do you want to Recompile?
 - Options for Recompilation?
 - What to Recompile?
- User-defined Functions
 - Functions v. Stored Procedures
 - Scalar functions for simple expressions
- Extra Info: Stored Procedure Best Practices

Stored Procedure Best Practices

- Many Best Practices well documented in whitepaper titled: *Query Recompilation in SQL 2000*
<http://msdn.microsoft.com/library/default.asp?url=/nhp/Default.asp?contentid=28000409>
- Naming Conventions
 - Owner Qualify
 - Do not use sp_
- Modifying Procedures
- Write Solid Code
 - Writing Better Queries/Better Search Arguments
 - Changing Session Settings
 - Interleaving DML/DDDL
 - Temp Table Usage
 - Modular Code
- Detecting Excessive Recompilations

Naming Conventions

- Owner Qualify to Eliminate Ambiguity
 - On execution
EXEC dbo.procname
 - On creation
CREATE PROC dbo.procname
AS
SELECT columnlist FROM dbo.tablename
EXEC dbo.procname
- Minimize Blocking – initial cache lookup by owner will fail. It will not cause a recompile but excessive lookups can cause significant blocking and cache misses.
- Do not use sp_ in stored procedure names – causes cache misses on lookup as well because SQL Server looks in master first!

See KB Article Q263889

Modifying Procedures

- **DROP and RECREATE**
 - Loses the dependency chain stored in sysdepends
 - Loses the permissions already granted
 - Invalidates all plans
- **ALTER PROC**
 - Loses the dependency chain stored in sysdepends
 - ❖ Retains the permissions
 - Invalidates all plans
- **To retain the dependency chain you must also ALTER all procedures that depend on the procedure being altered.**

Changing SESSION Settings

- **Certain Session Settings can be set within a stored procedure – some can be desired:**
 - SET NOCOUNT ON
 - SET QUOTED_IDENTIFIER OFF (not recommended except for backward compatibility and upgrades)
- **Some Session Settings will cause EVERY execution to force a recompile:**
 - ANSI_DEFAULTS
 - ANSI_NULLS (*tip: do not use WHERE col = null, use col IS NULL*)
 - ANSI_PADDING
 - ANSI_WARNINGS
 - CONCAT_NULL_YIELDS_NULL (*tip: use the ISNULL function to concatenate strings*)
- **Recommendation: DO NOT Change these session settings in the client or the server!**
See “SET Options that Affect Results” in the BOL

Interleaving DML/DDL Statements

- Objects that don't exist at procedure first execution cannot be optimized until statement execution
- Upon execution of a DDL statement the procedure gets recompiled to recompile the plans for the DML
- But wait – not all of the objects are created...so later executions of DDL force recompilation AGAIN...
- Don't interleave DDL and DML separate it...
- All DDL at the beginning of the proc, all DML later!

Data Manipulation

- **Derived Tables**
 - Nested Subquery in FROM clause
 - May optimize better than temp tables/variables
- **Views**
 - Another option – rewrite existing temp table code to use views instead (simple rewrite)
 - May optimize better than temp tables/variables
- **Temp Tables**
 - Should be considered
- **Table Variables**
 - Limitations might not affect you
 - Might be the most optimal

Temp Table Usage

- Temp Table can create excessive recompilations for procedures. Consider creating permanent tables (with indexes) and manipulating data there.
- Consider dropping and re-creating or rebuilding indexes as part of the procedure instead!
- Try not to create tables conditionally (IF create... ELSE create...)
- Use Profiler to see if there are significant recompiles
- Use KEEP PLAN on SELECT statements if data changes more than 6 times but the plan should not change.

Table Variable Usage

- Scope is limited to the local procedure\transaction
- Does not cause excessive recompiles due to local only access
 - No re-resolution on CREATE/ALTER
 - Temp Tables need re-resolution for nested procedures
- Only Key Indexes can be created
 - Definition of Table allows PRIMARY KEY/UNIQUE constraint indexes
 - Use TEMP TABLES if large volumes of data will be manipulated – create the right indexes for access
- Population
 - Does not support INSERT EXEC
 - Does not support SELECT INTO

Temp Table vs. Table Variables

- **Temp Table**
 - **PROs**
 - Can create useful nonclustered non-unique indexes to improve join performance
 - Can access from other nested procedures
 - Can populate with INSERT EXEC or SELECT INTO
 - **CONs**
 - Potential for excessive recompiles due to resolution
- **Table Variable Table**
 - **PROs**
 - Local only – no excessive recompiles
 - **CONs**
 - Cannot create additional nonclustered indexes
 - Not flexible on population

Detecting SP Recompilation

Event = SP:Recompile & Column = EventSubClass

1	Local Schema, bindings or permissions changed between compile and execute or executions Shouldn't happen often. If it does, isolate where/how changes occur and batch/sched. off hours
2	Statistics changed Thresholds for statistics of the different types of tables vary. Empty Tables (Permanent >= 500, Temp >= 6, Table Variables = No threshold) Tables with Data (Perm/Temp >= 500 + 20% cardinality, Table Variables = No threshold) If consistent plan then eliminate recompiles from changes in statistics by using (KEEPFIXED PLAN) optimizer hint in SELECT
3	Object not found at compile time, deferred check at run-time If the objects on which the procedure are based are permanent objects consider recreating
4	Set option changed in batch Best Coding practice: Consistency in client session settings. Consistency in development environment. Only use SET options when connection is started and when procedure is created.
5	Temp table schema, binding or permission changed Change coding practice for #temptable
6	Remote rowset schema, binding or permission changed. Gets stats from remote server, may recompile. If you're going to another server often – for a relatively small amount of static data you might consider periodically brining over a local copy?

Profiling SP Performance

- **Create New Trace (SQLProfilerTSQL_sps)**
- **Replace SP:StmtStarting w/SP:StmtCompletion**
 - **Better if you want to see a duration (starting events don't have a duration)**
 - **Add Duration as a Column Value**
- **If short term profiling for performance:**
 - **Add columns: Reads, Writes, Execution Plan**
- **Always use Filters**
 - **Database Name (only the db you want)**
 - **Exclude system IDs (checkbox on filter dialog)**