

# Database Upgrade to SQL Server 2008, Tools and Approaches

**Writer:** Dmitry Artemov

**Contributor:** Alexei Khalyako

**Technical Reviewer:** Alexei Khalyako, Kevin Cox, Mark Souza, Murshed Zaman, Peter Scharlock

**Editor:** Beth Inghram

**Published:** May 2010

**Applies to:** SQL Server 2008 R2

**Summary:** Performing an upgrade is a complex and often risky project. A successful upgrade can provide an organization with a modern platform for development and production. An unsuccessful upgrade can cause lost time and money, and it can create a bad perception for the future.

This paper documents approaches and tools that can help DBAs and developers to achieve successful and mostly painless upgrade of SQL Server databases from Microsoft® SQL Server® 2000 or SQL Server 2005 to SQL Server 2008 (or SQL Server 2008 R2).

In the methodology this paper discusses, a SQL Server workload is captured on customer production environment, and then this identical test workload is run on the old and new versions of SQL Server in a test environment to compare performance and isolate any problem queries. For this approach, we use components from the "RML Utilities for SQL Server" set of tools.

Copyright

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This white paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in, or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

© 2010 Microsoft Corporation. All rights reserved.

Contents

[Database Upgrade to SQL Server 2008, Tools and Approaches 1](#_Toc261262584)

[Introduction 5](#_Toc261262585)

[How long it may take to upgrade 6](#_Toc261262586)

[Planning 7](#_Toc261262587)

[Application compatibility check 7](#_Toc261262588)

[SQL Server Books Online as a resource 8](#_Toc261262589)

[Technical reference documents for upgrading 9](#_Toc261262590)

[Upgrade Advisor 9](#_Toc261262591)

[Testing the upgrade 9](#_Toc261262592)

[Scope 10](#_Toc261262593)

[Test environment options 10](#_Toc261262594)

[Loading application 13](#_Toc261262595)

[Tracing and trace replay in the test environment 13](#_Toc261262596)

[ReadTrace and how to make the most of it 14](#_Toc261262597)

[Preparing for tracing 14](#_Toc261262598)

[Tracing data storage considerations 18](#_Toc261262599)

[Collecting data 18](#_Toc261262600)

[Processing traced data 20](#_Toc261262601)

[Tables created during processing 22](#_Toc261262602)

[Cleaning up the RML files 22](#_Toc261262603)

[Preparing the test environment 24](#_Toc261262604)

[Converting trace files into load scripts 26](#_Toc261262605)

[Replaying traces in the test environment 32](#_Toc261262606)

[Parsing replay trace files 33](#_Toc261262607)

[Preparing data for analysis 35](#_Toc261262608)

[Analyzing test result differences 39](#_Toc261262609)

[Testing with database compression implemented 43](#_Toc261262610)

[Possible upgrade issues and workarounds 45](#_Toc261262611)

[Index schema changes and hints to the rescue 46](#_Toc261262612)

[Code rewrite 52](#_Toc261262613)

[Rewriting code when code is not available 55](#_Toc261262614)

[Conclusion 55](#_Toc261262615)

[References 55](#_Toc261262616)

# Introduction

Any organization at some point faces a necessity to upgrade a trusted and proven platform. There could be many reasons for this: end of vendor support, the inability to fully use the latest hardware, or business requirements that are hard or impossible to implement within the set of tools that the currently installed version provides. The list may go on and on.

This article concentrates on one part of the platform, Microsoft SQL Server. More specifically, it is about upgrading SQL Server 2000 or SQL Server 2005 to SQL Server 2008.To start, I suggest upgrading SQL Server 2000 databases directly to SQL Server 2008, rather than upgrading first to SQL Server 2005 and from there to SQL Server 2008.

So, why skip an intermediate upgrade to SQL Server 2005? There are several reasons:

* The SQL Server 2005 release took place more than four years ago, so the version has an accordingly shortened support period. Because SQL Server 2008 was released just one year ago, your organization will enjoy full support from Microsoft much longer.
* Upgrading from SQL Server 2000 to SQL Server 2005 requires approximately the same effort as upgrading from SQL Server 2000 to SQL Server 2008. You will ultimately upgrade to SQL Server 2008 or even future versions, so it is not necessary to upgrade in several steps or to perform what is essentially the same activity twice.
* SQL Server 2005 is no longer for sale; purchasing SQL Server 2008 is your only option (although you can request a downgrade license and media, if you need SQL Server 2005). If you are paying for SQL Server 2008, why not take advantage of it?

In terms of functionality, consider that SQL Server has historically had a pattern of releasing in pairs of major and minor versions. You can pair SQL Server 6.0 and SQL Server 6.5, SQL Server 7.0 and SQL Server 2000, and finally SQL Server 2005 and SQL Server 2008. The first (major) release of the pair usually brings new technologies, new functionality, architecture, and a change of the internals for SQL Server; the second (minor) release builds upon the first, expanding and leveraging what was implemented in the previous release. Thus, you can expect something really new in the next version of SQL Server, and changes could be so deep that migration would be very difficult if possible at all. So you should not wait to upgrade your SQL Server 2000 installation. Upgrade to SQL Server 2008 and get access to the richest and most advanced release that you can find as of today.

For more information about past and present of SQL Server, see "[The Evolution of Microsoft SQL Server: 1989 to 2000](http://insidesqlserver.com/companion/History%20of%20SQL%20Server.pdf) " (http://insidesqlserver.com/companion/History%20of%20SQL%20Server.pdf) or "[SQL MythBusters – "SQL Server is really a Sybase product not a Microsoft one](http://blogs.msdn.com/euanga/archive/2006/01/19/514479.aspx)." (http://blogs.msdn.com/euanga/archive/2006/01/19/514479.aspx).

# **How long it may take to upgrade**

Of course, any upgrade process should always include a planning phase. Before jumping into the upgrade process, the administrator or the manager of IT department should consider the amount of time the upgrade will take and the risks it may entail. Well, obviously there is no simple way to determine these factors, but I can share information about upgrades I’ve performed with different types of organizations:

* Project 1: A very big telecom data warehouse (approximately 25 terabytes of data) was preparing to move from SQL Server 2005 to SQL Server 2008.
* Project 2: A retail shop with relatively small amount of data (a couple of terabytes) was upgrading their replication infrastructure between HQ and regional shops from SQL Server 2000 to SQL Server 2008.
* Project 3: A large, mission-critical banking application with lots of business logic in stored procedures and no dependencies on "surrounding" technologies like replication, SQL Server Reporting Services, and DTS was upgrading from SQL Server 2000 to SQL Server 2008.

Project 1 took about three weeks of not-very-intensive preparation, including one week focused mainly on understanding performance gains of the new platform and testing the functionality. The application had small number of very complex queries, and the actual upgrade task was to properly implement new SQL Server 2008 functionality that enabled engine work with large amount of data. For more information, see the technical article [Using Filtered Statistics with Partitioned Tables](http://sqlcat.com/msdnmirror/archive/2009/10/20/using-filtered-statistics-with-partitioned-tables.aspx) (http://sqlcat.com/msdnmirror/archive/2009/10/20/using-filtered-statistics-with-partitioned-tables.aspx).

Project 2, however, took three months of very intensive work. The most difficult goal was to comply with a set of very strict limitations that the customer set in advance. The application in question was implemented on the 32-bit version of SQL Server 2000, working on a cluster and with very heavy use of transactional replication (170 subscribers, hundreds of articles). Poor connectivity required us to upgrade without replication resynchronization, and an upgrade to the 64-bit version of SQL Server 2008 made an in-place upgrade impossible. For more information about how we solved this puzzle, see [Upgrading Replication from SQL Server 2000 32-Bit to SQL Server 2008 64-Bit without re-initialization](http://sqlcat.com/technicalnotes/archive/2010/02/08/upgrading-replication-from-sql-server-2000-32-bit-to-sql-server-2008-64-bit-without-re-initialization.aspx) (http://sqlcat.com/technicalnotes/archive/2010/02/08/upgrading-replication-from-sql-server-2000-32-bit-to-sql-server-2008-64-bit-without-re-initialization.aspx).

Project 3 also took about three months. Here most of the time we spent fighting execution plans that changed when migrating from SQL Server 2000, introducing a much larger number of reads. For example, with several thousand queries in replay set, if 5 percent suffer from bad performance, that represents several dozen queries which must be analyzed and optimized (in this case mostly SELECTS involving 6 or more joins).

These examples may help you reach a very rough estimate, but SQL Server is a large and complex product, and every organization is different. As usual, the only thing we can say with certainty when discussing SQL Server is, "It depends."

# **Planning**

An upgrade is a full-scale project, and planning is one of the most important pieces of any project that affects your main data processing application. A properly prepared test environment and clearly defined goals will greatly help to reduce risks and enable you to gather reliable information that will result in a well-performing application, helping to achieve business goals for the organization.

When you plan the upgrade process, you should understand what you can and cannot expect from the upgraded application. First of all, do not expect miracles. Sometimes they do happen, but in general, upgrades are a lot of time and work. If you compare SQL Server 2000 and SQL Server 2008, you will see that that they are two different products. The fact that a custom application or a particular configuration worked in SQL Server 2000 does not guarantee that it will work the same way in SQL Server 2008. While most queries will perform the same or much better, some plans in SQL Server 2008 may change and lead to poor performance. When you decide to upgrade, consider all pieces of the environment in which the application works. You should note all aspects of client applications, functionality, as well as implementation of "peripheral" components, technologies used, and so on. When you plan the upgrade process, look at all aspects that may affect functionality and/or the length of time it takes to move application from one version to another. Include possible caveats on the way.

## Application compatibility check

First of all, determine whether anything in your application that might prevent an upgrade, such as Transact-SQL statements that are not supported in the version to which you are upgrading or any changes in the new version of SQL Server that might break existing features of your application.

For example, I know developers that used "sys" for user-defined schema. Now, starting from SQL Server 2005, "sys" is system-reserved schema name, and attempts to attach a database that contains a table named sys.systable will raise an error:



**Figure 1:** Failing to check compatibility issues may cause upgrade project to fail

Identify and correct these issues before you start the actual upgrade. For more information, see SQL Server Books Online, especially the “What’s New” section, and articles about performing upgrades.

Finally, if you plan to upgrade a database that is used by a third-party application, verify that the vendor supports SQL Server 2008. The vendor may also be able to provide support or helpful information about the upgrade process.

## SQL Server Books Online as a resource

It is necessary to carefully read articles in SQL Server Books Online that describe breaking changes, functionality that is no longer supported, and functionality that will soon become unsupported. For SQL Server 2008, SQL Server Books Online mostly describes differences between SQL Server 2008 and SQL Server 2005; additionally, if you are upgrading from SQL Server 2000, you should read SQL Server Books Online for SQL Server 2005, especially where it describes differences between SQL Server 2000 and SQL Server 2005. Those two versions of SQL Server Books Online will provide a complete picture of the differences between SQL Server 2000 and SQL Server 2008. For example, for more information about backward compatibility issues in the database engine between SQL Server 2008 and SQL Server 2000, see [SQL Server 2005 Database Engine Backward Compatibility](http://technet.microsoft.com/en-us/library/ms143532%28SQL.90%29.aspx) (http://technet.microsoft.com/en-us/library/ms143532(SQL.90).aspx) and [Backward Compatibility](http://technet.microsoft.com/en-us/library/cc280407.aspx) (http://technet.microsoft.com/en-us/library/cc280407.aspx).

## Technical reference documents for upgrading

Two large documents (350 pages each) titled “SQL Server 2008 Upgrade Technical Reference Guide” and “SQL Server 2005 Upgrade Technical Reference Guide” should be read as part of the upgrade planning process. You can use these two white papers to identify potential issues you may encounter with components and technologies that your application currently uses. For links to these guides and other useful documents, see “References” at the end of this white paper.

## Upgrade Advisor

Then download and install Microsoft SQL Server 2008 Upgrade Advisor. You can find this tool on your distribution media, but downloading it from the Microsoft site will provide you with the most up-to-date version, which contains improved analysis and better rules. This tool automates detection of incompatibilities. It can find deprecated syntax constructs and determine keywords used in database schema, and it should be used against existing databases. Upgrade Advisor is useful, but again, don't expect miracles.

If the client application contains Transact-SQL code, consider getting SQL Server Profiler traces and feeding them to Upgrade Advisor. However, bear in mind that processing large traces can be time-consuming because of the rules Upgrade Advisor must check.

If Upgrade Advisor finds anything, it reports it and provides information on the object or objects in question and the severity of the issue. Based on this information, you can make a decision to proceed with upgrade, or you might decide to perform further testing. Later in this article, I describe a method that enables you to compare performance of the same code under two different versions of SQL Server.

## Testing the upgrade

After all checks are done, you can start the actual upgrade in a test environment. The key word here is "test". One of the features that make SQL Server applications very difficult to tune is its uniqueness. The same code may perform differently on different amounts of data or different hardware configurations. So “Test” should be your motto when you approach SQL Server upgrades. The more closely you are able to imitate your production environment, the fewer surprises you will have when you go live with the upgraded application.

## Scope

Testing generally focuses on two areas: functionality and performance. Usually functionality tests are easier to set up because they are supported by existing tests scripts or other test means. The value of functionality testing is quite obvious: Unless you can test business functionality, you cannot send an application to real life. Setting up stress tests is harder, and many of my customers unfortunately do not build proper stress-test environments. Neglecting to perform rigorous stress testing can lead to some very unpleasant surprises when you upgrade an application.

Here’s one example from my personal experience: I was involved in a project where we upgraded a very complex application that relied heavily on transactional replication. We built an upgrade plan and created a test environment, and we were able to properly test all upgrade steps where, as we expected, we found the biggest risk. When we did the actual production upgrade, the steps that we feared the most went without failure and on schedule. But when the system started, we found that of the expected 1,500 SQL Server Agent jobs, only 400 were running. As it turned out, SQL Server 2008 has different (and not always clearly documented) settings for SQL Server Agent subsystems. In our test environment we didn't plan for this, and we didn't even create test jobs to see whether that many jobs could run at the same time. We had to open a critical severity case with premier support and finally managed to set SQL Server Agent properly, but it cost us 4 hours of overtime. What is the lesson? It's simple: The test environment and the test load should be as close as possible to the actual production environment and expected stress levels.

## Test environment options

You have several options when planning your test environment. In an ideal world, the development team should have actual scripts that were created in some professional tool like Microsoft Visual Studio® or Load Runner. Those scripts enable you to test all aspects of application functionality with different loads, various numbers of users, various combinations of operations, and so on. But in real life, building such scripts and keeping them synchronized is not an easy or cheap task, and you may find that all you have is some Transact-SQL scripts that just cover basic functionality, not parameterized, that cannot be run simultaneously. What you can do if you find yourself in this situation?

You should create a test environment, from scratch if necessary, using what is available, and do it quick:

1. Check what you have, collect scripts and homegrown tools, and see how you can use them.
2. Determine what areas of functionality existing tools cover.
3. Work with your development and business teams to find out how you should set parameters when running scripts, what the upper and lower limits are for those parameters, and how all those pieces of tests should work together.

For Transact-SQL code that you plan to use, you can create wrappers that enable you to call tested code and assign parameters to it. It is pretty simple. In more complex situations when two pieces of a test should be run in some interdependent way, the wrapper would require a little more code, but it is still doable.

Here’s an example.

CREATE TABLE Wrapper1\_ParamSet (

 (paramId INT ,

 Param1\_Value DATE ,

 Param2\_Value CHAR (10)

);

INSERT Wrapper1\_ParamSet

VALUES (1, '20080101', 'Param\_a1')

INSERT Wrapper1\_ParamSet

VALUES (2, '20080202', 'Param\_a2');

INSERT Wrapper1\_ParamSet

VALUES (3, '20080303', 'Param\_a3');

INSERT Wrapper1\_ParamSet

VALUES (4, '20080404', 'Param\_a4');

GO

CREATE PROCEDURE Wrapper1

AS

DECLARE @Parm\_Num AS INT;

DECLARE @parameter1 AS DATE;

DECLARE @parameter2 AS CHAR;

-- Use RAND to get a random ID for the set of parameters

SET @Parm\_Num = CAST (((4 - 1 + 1) \* RAND() + 1) AS INT);

-- Collect the actual values of the parameters

SET @parameter1 = (SELECT param1\_value

 FROM Wrapper1\_ParamSet

 WHERE paramId = @Parm\_Num)

SET @parameter2 = (SELECT param2\_value

 FROM Wrapper1\_ParamSet

 WHERE paramId = @Parm\_Num);

-- Now run the test query with the parameters

SELECT \*

FROM

pubs..sales

WHERE ord\_date = @parameter1

 AND payterms = @parameter2

--Now you can call Wrapper1 from the loading application.

Execute Wrapper1

## Loading application

Wait… What loading application? A loading application is an application capable of opening as many connections to SQL Server as needed and sending EXECs, SELECTs, and other commands over those connections. Some advanced tools can set delays between database calls, run sets of jobs with different settings, and so on. This paper discusses the OSTRESS utility from RML Utilities. You can also go search www.codeplex.com or sourceforge.com for free tools for download. If the tool cannot run several differently configured sessions, run several instances of the tool and configure them accordingly.

If your existing set of scripts does not cover your testing needs, you can use server traces to build your test environment. This process is discussed in the remaining portion of this white paper.

# **Tracing and trace replay in the test environment**

If you have earlier experience with SQL Server Profiler, you should know that the tool can not only record production database activity, but also replay it on test server to imitate actual load. I tried to use SQL Server Profiler replay with mixed results. First of all, it can consume all 100 percent of your workstation processor, thus making it useless for any other activity like database server monitoring. Next, sometimes it simply refuses to read trace (.trc) rollover files. This is a big problem, because for even a reasonably loaded system you will have many trace files (on my last engagement we got about 500 GB in more than one thousand trace files). Of course, you can create a single huge trace file but it would be much better to use RML Utilities for driving the replay. This is a free download that you can find on the Microsoft site by searching for "RML Utilities for SQL Server". Here is what you can do with the trace files:

* Very quickly parse trace files and load them into the database (this requires a minimum of SQL Server 2005). *Quickly* means that 1 GB of traces can be parsed within one minute, if you properly set up the parsing process (ReadTrace settings are discussed in greater detail later in this paper).
* Convert trace files into load scripts that can be replayed against the test server.
* Use the database that accepts trace file contents for comparative performance analysis. This enables you to determine which portions of application code can cause problems after upgrade. It can also be very useful for creating a baseline for your production server.

## ReadTrace and how to make the most of it

RML Utilities for SQL Server includes two main pieces that I use in upgrade projects: ReadTrace and OSTRESS. ReadTrace parses trace files and produces load scripts, and OSTRESS is a replay (loading) utility.

### Preparing for tracing

Before you can use ReadTrace, you have to build a trace session and collect trace files. It is important to note that you should use a server-side trace, because this type of trace uses fewer resources than SQL Server Profiler. This means that you will have to build Transact-SQL code that defines and starts the trace session. A careful look at the trace definition shows that it consists mostly of stored procedure calls that look like this.

exec sp\_trace\_setevent @TraceID, 10, 1, @on

exec sp\_trace\_setevent @TraceID, 10, 3, @on

exec sp\_trace\_setevent @TraceID, 10, 6, @on

exec sp\_trace\_setevent @TraceID, 10, 9, @on

And so on, dozens and dozens of such lines. Obviously writing such code manually is not an easy task, because you have to remember those event IDs. But thankfully you can use SQL Server Profiler itself to generate most of the trace definition script.

You can open the SQL Server Profiler GUI, set necessary events and the data columns for them, and save resulting definitions as Transact-SQL scripts and/or trace templates. I would recommend doing both, because you can reuse traces from templates. The RML Utilities Help file (RML Help.pdf) describes the necessary events and their data in the section titled "Required Trace Events."

The script from SQL Server Profiler requires some minor changes that set proper trace file names and point the output to the proper folder. These changes are easy to make. Here is the sample of a script header.

-- Trace definition script

/\*-----------------Header starts ------------------------------------\*/

declare @rc int

declare @TraceID int

declare @maxfilesize bigint

set @maxfilesize = 250 -- single trace file size in MB

declare @currdate datetime

declare @filename nvarchar(100)

declare @stoptime datetime

--- assign value for the @elapsed variable to set tracing duration

declare @elapsed smallint

declare @filescount int -- how many files we want to keep (sliding window)

set @elapsed = 60 -- trace will automatically stop after @elapsed minutes

set @currdate = getdate()

set @stoptime = dateadd(minute, @elapsed, @currdate)

select @filename = convert(datetime, @currdate, 112)

select @filename = replace(replace(@filename, ' ','\_'),':','-')

-- Correct @filename

-- in order to use proper folder

select @filename = 'D:\temp\' + @filename

--select @filename

exec @rc = sp\_trace\_create @TraceID output, 2 /\* use rollover\*/, @filename, @maxfilesize, @stoptime, NULL /\* keep all TRC files \*/

if (@rc != 0) goto error

/\*----------------Header ends ------------------------------\*/

-- Next goes SQL Server Profiler generated code.

**Note:** You can use this code to create any type of trace, not just those for use during an upgrade but for any tuning, development, or other task that will require detailed profiling of an application.

If you need to create a trace that will serve as a base for performance investigation in an upgrade project, and if you plan to use ReadTrace to parse trace files, consider using the TraceCaptureDef.sql script that’s included in the RMS Utilities samples.cab file (installed in the \RMLUtils\Samples\ folder).

This script determines the version of SQL Server being used and chooses sets of events accordingly. All you have to do is to add the header described earlier.

Open TraceCaptureDef.sql in any text editor and find the following code, inserting the header as shown.

 exec sp\_trace\_setevent @TraceID, @iEventID, @iColID, @on

 set @iColID = @iColID + 1

 end

end

go

-- Place header here

/\*-----------------Header starts ------------------------------\*/

-- Header goes here

/\*-----------------Header ends --------------------------------\*/

-- Here goes the rest of TraceCaptureDef.sql code

declare @off bit

Take the following information into account when you build file names:

* Don't use Cyrillic letters in file names (the initial version of this white paper was written for a Russian audience). ReadTrace converts Cyrillic into unreadable characters, which complicates diagnostics. It works, but your log will be difficult to decipher.
* Do not end file names with a digit. ReadTrace expects to find incremental numbering at the end of the file name. The server adds numbering itself, like this: TraceFile.TRC, TraceFile1.TRC, and so on.
* Use the same version of ReadTrace across all tests. Methods described here rely heavily on the HashId that ReadTrace generates for the Transact-SQL code, and it can be the same across tests only if you use the same version of ReadTrace to parse trace files.

Before going further, I'd like to mention that you can extend the trace definition with two event classes that were introduced in SQL Server 2005: the **Deprecation Announcement** event class and the **Deprecation Final Support** event class. To add these events to the trace definition, run SQL Server 2008 Profiler, connect to an instance of SQL Server 2008, click the **Events selection** tab in the Trace properties window, and then select the **Show all events** and **Show all columns** check boxes. Clear the check boxes for all events selected by default, expand the Deprecation event group, and then select the **Deprecation announcement** and **Deprecation final support** check boxes. Run and immediately stop the trace. On the **File** menu, point to **Export**, point to **Script trace definition**, and then click **For SQL Server 2005-2008,** and then save the trace definition to an SQL file. Cut all commands that have an EXEC **sp\_trace\_setevent** call and paste them into the trace definition that you used earlier. Tracing deprecated events will help you diagnose potential issues that may be caused by old syntax in your code.

When the trace definition script is ready, try to run it for several minutes under normal production server load, and then parse the resulting trace file. It will help to determine whether you have all of the necessary events defined. If something is missing, ReadTrace writes a warning in the log and on the console. This warning is displayed when the first trace file is loaded, like this:

10/19/09 21:29:00.674 [0X00000B68] WARNING: The 'EventSequence' column is missing from the trace. Physical file order will be used and this can lead to inaccurate results.

Your trace places additional load on the production server. The trace definition should include the most detailed trace events and execution plans in text form, which causes additional load, so your users may complain about application performance.

**Important:** Immediately before you begin the trace, be sure to create a full database backup, and use it for all testing activity. Otherwise during replay you may find that you are trying to enter data that is already present and get a primary key violation error.

### Tracing data storage considerations

How much data should you expect when tracing a production system? Prepare for a lot. As mentioned earlier, on a reasonably loaded system, you can easily get hundreds of gigabytes for several hours of tracing. I got up to a gigabyte per minute. Some additional considerations should help you to reduce stress on a production system:

* Consider using a folder that is on a disk not used by SQL Server; it is not a good idea to overload the I/O subsystem where the database files reside.
* If you have antivirus software, tell it not to check \*.trc files.

In order to reduce disk space consumption, you can do the following:

* Create a compressed folder for trace files if you have extra processor power. Windows® compression can reduce file size several times. Note that if your database server uses SAN, you most probably have formatted logical drives with a block size of 64К, which disallows compression.
* Try an external USB drive, but make sure that is has enough capacity and that your server has USB 2.0 slot. It is best to use an empty, freshly formatted drive. An empty drive provides maximum performance, and file fragmentation will not affect data transfer.

### Collecting data

After you test the trace process, you can start actual data collection.

Use the following command file to run the process.

OSQL -E -S serverName\InstanceName -Q "Backup database Main to disk='D:\Backup\MainToTrace.bak"

REM If you upgrade SQL Server 2005 database, use WITH COPY\_ONLY for backup

REM it will keep the normal backup chain that you use intact

REM As soon as a copy is created, you can start profiling

OSQL -E -S serverName\InstanceName -i "d:\CreateProfilerTrace.sql" -o "d:\CreateProfilerTrace.out"

CreateProfilerTrace.sql is the trace definition script, and CreateProfilerTrace.out is the output file where TraceID will be written (you need this trace id in order to stop tracing manually).

You also can collect data about running traces with the following command.

SELECT \* FROM ::fn\_trace\_getinfo(default)

For SQL Server 2000, this command returns information about a single trace. For SQL Server 2005, it returns data about two traces, unless you stopped default tracing.

As discussed earlier, the stop time is defined in the trace definition header, so under normal conditions the trace stops automatically. If you need to stop the trace manually, however, use one or both of the following two calls to the SQL Trace stored procedure **sp\_trace\_setstatus**.

exec sp\_trace\_setstatus 2, 0

-- stops tracing, but keeps the session, which can be restarted with status "1"

exec sp\_trace\_setstatus 2, 2

-- stops the session completely, closes the trace file, and removes the session

-- before setting status to 2 you should stop the trace with status of 0

The first parameter is the trace session ID that was written into the OUT file. The second parameter is the status ID that commands session behavior.

### Processing traced data

When tracing is over, it is necessary to get the trace files from the server and process them. As discussed earlier, you should have SQL Server 2005 or SQL Server 2008 available to create the database where ReadTrace will put the parsing results.

If you have many files and want to compact them, you can create a single huge ZIP archive and pass it to ReadTrace, which can read directly from ZIP. However, it cannot read from multiple archives. A much better approach would be to use WinRAR, which can compress every trace file into a separate RAR archive. In one case I worked on, a trace file of 350 MB was reduced to 350 KB. Such greatly reduced files are much easier to copy over the network, and the amount of time it takes to archive and open files on the server where you plan to process them is more reasonable.

Here’s a screen capture of the WinRAR interface.



Figure 2: WinrRAR can compress TRC files into separate archives.

So, copy files over and then extract them; now you are ready to parse traces and replay traced events. It should be noted that using trace files to set up a test session has some drawbacks. The most important one is that you can replay only what you trace. Everything else should be tested some other way. For example, you might need an important report that is run once a year. However, if you plan well ahead, you can set several tracing sessions and collect several sets of traces. You can also work with your development team to prepare reports as set of Transact-SQL commands that can be run separately or in parallel with trace replay (depending on how you run reports in production).

Now use ReadTrace to parse the trace files you collected. Before, I mentioned "correct" parsing settings. Now it's time to give more detailed explanations. For example, a command line could look like this.

ReadTrace.exe -SServer\Instance -E -I"C:\Temp\SQL2008\_test.trc" -f -T18 -T22 -dDb2008 -oTest2008

In this command:

* ReadTrace.exe - the parsing utility itself.
* -SServer\Instance - the name of the server and instance where ReadTrace creates a database in which to put parsed traces (the database is created if it does not exist; objects are dropped and re-created if the database does exist).
* -E - use Windows authentication (this is the default), but you can use the pair -U<user name> -P<password> instead.
* -I"C:\Temp\SQL2008\_test.trc" – the name and the full path for the first of the trace files.
* -f - a parameter that prevents the creation of replay files (RML). Do not use this parameter for the initial parse; it is used later in the process.
* -T18 – a trace flag that prevents Reporter from running. By default, when parsing is over, ReadTrace starts Reporter to display collected data, so it’s not necessary to start Reporter at this point.
* -T22 - By default, ReadTrace calculates aggregates for traced events that include averages, minimums, and maximums for reads, writes, CPU, and duration. Using this functionality can greatly increase parsing time, and it is usually faster to run a SELECT command against the trace database. So I recommend using this trace flag to prevent aggregation.
* -dDb2008 – the name of the database where ReadTrace will store parsed data.
* -oTest2008 – the name of folder where ReadTrace will write the execution log.

The actual command will look like this.

ReadTrace.exe -SServer\Instance -E -I"C:\Temp\SQL2008\_test.trc" -T18 -T22 -dDb2008 -oTest2008

Note that you don't use the -f key, because you do need the RML files to replay later.

### Tables created during processing

You may notice additional tables when processing is complete. Some of these tables are discussed in greater detail later in this paper, but they are worth mentioning here as well. When ReadTrace processes trace files, it produces two sets of tables. The first set consists of tables that store data for unique batches and data for all individual batches found in trace files. The unique batch table is called **ReadTrace.tblUniqueBatches**, and the individual batch table is called **ReadTrace.tblBatches**. The second set keeps information about unique statements and all individual statements accordingly. These tables are named **ReadTrace.tblUniqueStatements** and **ReadTrace.tblStatements**. These sets of tables are described in greater detail later in this paper.

Also, for tracing as detailed as that performed for the update process, ReadTrace creates tables that contain information about all of the login names, connections, parsed plans, unique application names found in traces, and other data, including information about the trace itself, like file names that were processed, events that were found in trace files, and so on. If you configure ReadTrace to calculate aggregates, ReadTrace creates tables such as **ReadTrace.tblBatchPartialAggs** and **ReadTrace.tblStatementPartialAggs.** These tables capture statistics about duration, reads, writes, and CPU for all unique statements and batches.

Finally, when parsing traces, ReadTrace creates a single RML file for each SPID recorded during tracing. This RML file is an XML file that contains all execution history for particular SPID. It is discussed in greater detail later in this paper.

### Cleaning up the RML files

When parsing is done, you may want to check to see whether you really need all of the RML files created during processing. To review this information, look at the parse database table [ReadTrace].[tblConnections].

For example, the following SELECT query might return something like the result set listed in the table.

SELECT SPID, ApplicationName, LoginName FROM [ReadTrace].[tblConnections] ORDER BY ApplicationName

|  |  |  |  |
| --- | --- | --- | --- |
| SPID | **ApplicationName** | **LoginName** | **Ignore** |
| **59** | **DTS DESIGNER** | **MON\_U** | **Yes** |
| **171** | **JSQLCONNECT** | **Domainname\UserName** | **Yes** |
| **59** | **MICROSOFT OFFICE 2003** | **User\_One** | **Yes** |
| **366** | **MS SQLEM - DATA TOOLS** | **Windowsuser1** | **Yes** |
| **134** | **OSQL-32** | **ServerName\HP ITO ACCOUNT** | **Yes** |
| **203** | **SQL QUERY ANALYZER** | **Windowsuser1** | **Yes** |
| **167** | **SQL QUERY ANALYZER - OBJECT BROWSER** | **Windowsuser1** | **Yes** |
| **137** | **SQL SERVER ENTERPRISE MANAGER** | **Windowsuser1** | **Yes** |
| **53** | **SQLAGENT - ALERT ENGINE** | **Domainname\MASTERSQL\_NIKON** | **Yes** |
| **52** | **SQLAGENT - GENERIC REFRESHER** | **Domainname\MASTERSQL\_NIKON** | **Yes** |
| **249** | **SQLAGENT - JOB MANAGER** | **Domainname\MASTERSQL\_NIKON** | **Yes** |
| 70 | My Business Application | User1 | **No** |
| 59 | My Business Application | User2 | **No** |
| 181 | My Business Application | User3 | **No** |
| 59 | My Business Application | User4 | **No** |
| 296 | My Business Application | User5 | **No** |
| 60 | My Business Reporting | User6 | **No** |
| 289 | My Business Reporting | User7 | **No** |

The SPIDs in bold type are of no interest for the upgrade process. Accordingly, you can safely delete SQL00059.rml, SQL00171.rml, and so on.

### Preparing the test environment

After you complete RML cleanup, start preparing your test server. You need both the current (for example, SQL Server 2000) and the new (SQL Server 2008) versions of SQL Server installed. Of course, your test installation should conform to all relevant best practices. On both versions, restore the production database backup that you created right before you started the trace.

When the database is restored, you may want to run a statistics update (with FULLSCAN if possible) and an index defrag.

To update statistics, you can use **sp\_updatestats**, but you cannot configure it to run with the full scan option. With UPDATE STATISTICS, you can use FULLSCAN, but it has to be called individually for all tables. Because neither option quite works for the upgrade scenario, I’ve created a script that creates the necessary commands.

Run one of the following scripts. These scripts generate a set of UPDATE STATISTICS commands. (These scripts should be run in the context of the upgraded database.)

For SQL Server 2005

SELECT 'update statistics [' + ss.name + '].' + '[' + object\_name(si.object\_id) + ']' +

' (' + si.name +')' + ' with FULLSCAN'

FROM sys.indexes AS si

 INNER JOIN

 sys.objects AS so

 ON si.object\_id = so.object\_id

 INNER JOIN

 sys.schemas AS ss

 ON so.schema\_id = ss.schema\_id

WHERE si.object\_id > 50000

 AND si.name IS NOT NULL;

For SQL Server 2000

SELECT 'update statistics [' + ss.table\_schema + '].' +

'[' + object\_name(si.id) + ']' +

 ' (' + si.name + ')' +

 + ' with FULLSCAN'

FROM sysindexes AS si

 INNER JOIN

 sysobjects AS so

 ON si.id = so.id

 INNER JOIN

 INFORMATION\_SCHEMA.tables AS ss

 ON so.name = ss.table\_name

WHERE si.id > 50000

 AND si.name IS NOT NULL

 AND so.xtype = 'U'

Copy and paste the generated script from the query results pane into the Transact-SQL editor, and then run it in context of the database you plan to upgrade.

In my experience, defragmenting indexes is necessary because SQL Server 2005 and SQL Server 2008 can have greatly reduced performance on fragmented indexes, even if the code does a plain primary key search. (I observed a situation in which a command like SELECT Fld21, Fld2, Fld3 FROM SomeTable WHERE PrimaryKey = 123 ran for 2 to 3 seconds before defragmentation and 10 milliseconds after.)

To defragment indexes, you can use a script provided in SQL Server Books Online. For more information, see section D, “Using sys.dm\_db\_index\_physical\_stats in a script to rebuild or reorganize indexes” in the Examples section of [sys.dm\_db\_index\_physical\_stats](http://msdn.microsoft.com/en-us/library/ms188917.aspx) (http://msdn.microsoft.com/en-us/library/ms188917.aspx).

**Note:** You can also use this script in production, to save time and resources; sometimes it may make more sense to use REORGANIZE, instead of using REBUILD for every index.

### Converting trace files into load scripts

In order to use RML as loading script, take a closer look at the RML files' internals. RML is a special type of XML. If you open an RML file in XML Notepad (another free tool in our toolbox), you will see that it contains all the information OSTRESS needs to replay recorded sessions against a test server.

RML contains CONNECT and sometimes DISCONNECT blocks and a set of LANG and/or RPC blocks. The CONNECT block contains all the information necessary to establish connections. The DISCONNECT block tells OSTRESS to close the connection, and the LANG/RPC block contains the executed code with transactions, parameterization, and so on.

Here’s an example of RML.

<?xml version="1.0" encoding="utf-16"?>

<RML>

<!-- TL: '2009-08-20 12:21:51:667' -->

<CONNECT SEQ="267049542" NEXTSEQ="267049543" DELTA="118175170000">

 <CMD>-- network protocol: TCP/IP

 set quoted\_identifier off

 set implicit\_transactions off

 set cursor\_close\_on\_commit off

 set ansi\_warnings off

 set ansi\_padding off

 set ansi\_nulls off

 set concat\_null\_yields\_null off

 set language us\_english

 set dateformat mdy

 set datefirst 7

 </CMD>

 <USER>Username\_LS</USER>

 <APPNAME>Application name v05.010.00</APPNAME>

 <NETWORK>TCP/IP</NETWORK>

 <DBID>7</DBID>

 <PACKETSIZE>4096</PACKETSIZE>

</CONNECT>

...

<LANG SEQ="267049603" NEXTSEQ="267049606" REPLAY\_DURATION="1" DELTA="118175270000">

 <CMD>use DepoMos</CMD>

</LANG>

<LANG SEQ="267054433" NEXTSEQ="267054453" REPLAY\_DURATION="1" DELTA="118177370000">

 <CMD>select USER\_NAME()</CMD>

</LANG>

<LANG SEQ="267054472" NEXTSEQ="267054485" REPLAY\_DURATION="1" DELTA="118177370000">

 <CMD>begin tran</CMD>

</LANG>

<LANG SEQ="267056449" NEXTSEQ="267057061" REPLAY\_DURATION="400000" DELTA="118178470000">

 <CMD> SELECT dbo.t\_Group.n\_G\_Id,

 dbo.t\_Group.s\_G\_Name,

 dbo.t\_Group.s\_G\_Describe

 FROM dbo.t\_User

 JOIN dbo.t\_UserGroup ON dbo.t\_User.n\_u\_id = dbo.t\_UserGroup.n\_u\_id

 JOIN dbo.t\_Group ON dbo.t\_Group.n\_g\_id = dbo.t\_UserGroup.n\_g\_id

 WHERE dbo.t\_User.n\_u\_id = CAST(22365 AS INT) and

 dbo.t\_Group.b\_G\_Deleted = 0

</CMD>

</LANG>

...

<LANG SEQ="267065411" NEXTSEQ="267073603" REPLAY\_DURATION="4400000" DELTA="118183370000">

 <CMD>commit tran</CMD>

</LANG>

...

<RPC SEQ="267241961" NEXTSEQ="267241966" REPLAY\_DURATION="1" DELTA="118277570000">

 <CMD>{call sp\_cursoropen(?, ?, ?, ?, ?)}</CMD>

 <PARM ORD="1" DECLARED="TRUE" TYPE="INT" INPUT="TRUE" OUTPUT="TRUE">

 <NAME>@cursor</NAME>

 <VAL>1243116</VAL>

 <OUTVAL>180150000</OUTVAL>

 </PARM>

...

</RPC>

...

<!-- TL: '2009-08-20 12:21:51:667' -->

<DISCONNECT SEQ="309664347" NEXTSEQ="9223372036854775807" DELTA="137560570000">

</DISCONNECT>

</RML>

For more information about RML format, see the RML Utilities Help file (RML Help.pdf). This section addresses some of the issues that seem to occur most frequently.

Let's start with the CONNECT block.

In order to run these tests, OSTRESS has to connect to the database. RML stores connection information in the CONNECT block. If all of your users work from domain accounts and therefore use Windows authentication, all you need to do is create accounts in a test environment (as domain or local users). You don’t need to edit the RML.

However, if your application uses SQL Server authentication, you will need to make some changes in RML file to adapt it to SQL Server authentication. This is because the OSTRESS tool that you will use to run these scripts can use only one SQL Server password at a time. SQL Server user passwords are unknown, and you may have more than one SQL Server account. There are several ways to adapt the scripts for SQL Server authentication:

* Edit the RML files and place the additional information in the CONNECT block.
* Use the command line parameter -a{password}. OSTRESS will use the user name from the CONNECT block and the password from a command prompt.
* Use several OSTRESS instances, each using a subset of RML files where all RMLs in a subset should be run under the same user account. In this case, use the -U{user name} and -P{password} command-line parameters.
* Force single user with -U{user name} and -P{password} command-line parameters; all connections will run under the account you pass from the command line.

In the last case, OSTRESS raises the following message after several attempts to use the user ID from the RML file:

02/09/10 11:17:04.110 [0x0000058C] WARNING: Multiple attempts to connect using the credentials specified in file c:\Program Files\Microsoft Corporation\RMLUtils\SQL00132-1.rml, global sequence 82 have failed; reverting to the connection credentials specified on the command line in order to continue processing.

In one case, I could not limit use to a single user name because the application was using the role-based security model and required actual user names. So we decided to edit the RML, explicitly placing the password into the CONNECT block.

<CONNECT SEQ="162410939" NEXTSEQ="162411012" DELTA="69096570000">

 <CMD>-- network protocol: TCP/IP

 set quoted\_identifier on

 set implicit\_transactions off

 set cursor\_close\_on\_commit off

 set ansi\_warnings on

 set ansi\_padding on

 set ansi\_nulls on

 set concat\_null\_yields\_null on

 set language us\_english

 set dateformat mdy

 set datefirst 7

 </CMD>

 **<USER>User name</USER>**

 **<PWD>User password</PWD>**

 <HOST>MON-A16006</HOST>

 <APPNAME>DTS Designer</APPNAME>

 <NETWORK>TCP/IP</NETWORK>

 <DBID>7</DBID>

 <PACKETSIZE>4096</PACKETSIZE>

</CONNECT>

Now imagine that you have several hundreds of RMLs (like I did in my last project). In this case, editing each file would not be an easy task. You can use a text search-and-replace tool instead of making the change manually. (I used one called Text Workbench, which I downloaded from <http://www.silveragesoftware.com/hffr.html>.) What we did was to insert a <PWD>User password</PWD> block into RML, and we used the same password for all users.

Because you can use a single password, you can assign it to all SQL Server logins. If you have many logins, you can generate a set of commands by using the following script.

-- Should be run in the context of the trace database

SELECT DISTINCT (loginname)

INTO tempdb..Logins

FROM ReadTrace.tblConnections

SELECT 'execute sp\_password @new=''' + 'pass@word1' +

'''' + ',' + '@loginame=' + '''' + loginname + '''' +

CHAR(13) + '' + CHAR(13)

FROM tempdb..Logins

This code returns a set of commands, which you then run against the **master** database on the test server for both instances of SQL Server. This step is also necessary if you decide to use the -a{password} command-line parameter.

Also, you may find that the CONNECT block contains a DBID value. In this case, OSTRESS may generate an error like this:

02/09/10 10:53:40.891 [0x00000F5C] ERROR: CONNECT node (SEQ=82) specifies <DBID>7</DBID> but no database mapping was found for this database. Refer to the help file sections for dbid\_map.txt and/or trace flag -T33 for details on how you can modify this behavior.

This error means that OSTRESS could not map the DBID to an actual database. To address this issue, create a text file called dbid\_map.txt and put mapping information in the form of <DBID>,<DATABASE NAME> pairs in it. Also add trace flag -T33 to the command line. Another option is to edit the RML files and replace the DBID tag (in this case, <DBID>7</DBID>) with the DATABASE tag (<DATABASE>DatabaseYouAreTesting</DATABASE>).

This is almost all that may be necessary to prepare RMLs. Of course, you can make more sophisticated edits to overcome OSTRESS limitations. For more information about these limitations, see "Constructs that May Not Replay" in the RML Utilities Help file (RML Help.pdf).

**Note:** One such limitation is multiple active results set (MARS) support. For more information about how to address this limitation, see [RML Utilities - ReadTrace and how to workaround MARS](http://blogs.msdn.com/psssql/archive/2009/01/21/prb-rml-utilities-readtrace-and-how-to-workaround-mars.aspx) (http://blogs.msdn.com/psssql/archive/2009/01/21/prb-rml-utilities-ReadTrace-and-how-to-workaround-mars.aspx).

After all the preparation steps are done, you should back up your test database in order to have it fully prepared and ready for subsequent tests. You may need to repeat the test several times; ideally, you should keep a master copy. When you back up a SQL Server 2008 version of the test database, use backup with compression. You will be pleasantly surprised at the quickness and compression factor of the backup. This is a byproduct of the larger testing process, but it will enable you to compare backup performance in your current version with the newest version of SQL Server.

## Replaying traces in the test environment

When your RMLs are ready, you can start the replay process.

First, replay RMLs against the SQL Server 2000 database, and then replay them against the SQL Server 2008 database. In both cases, with the same script you used to collect traces from the production server, collect a fresh set of traces from the test server. Those trace files provide you with data to compare server behavior.

**Note:** Don't forget to shut down unnecessary instances of SQL Server before you start the next test.

The call to OSTRESS looks like this.

OSTRESS -SServer\sql2000 -UUser -PPassword -dDatabase -i"G:\RML\\*.rml" -oc:\temp\ReadTraceOutput -mreplay -cOSTRESS.ini -T146

The parameters are pretty self-explanatory. For more information about the control file and the –c parameter, which determines the control file, see the "OSTRESS" section in the RML Utilities Help file (RML Help.pdf). You may also want to consider using trace flag 146 when replaying against SQL Server 2000, to avoid an attempt to set an encrypted connection, which fails on SQL Server 2000. For more information, see "Encrypted Connections" in the RML Utilities Help file.

When you have run all tests and collected the new set of traces, you can compare performance.

You have to parse two sets of trace files (for your current version of SQL Server and SQL Server 2008) and compare how code behaves.

## Parsing replay trace files

The trace files that you have will be processed the same way that the RML files were processed. The only difference here is that you need to use the –f key, because this time there is no need for two sets of RML files.

At the end of processing, you will have two databases filled with useful information.

In ReadTrace, when a trace file is processed, all command text is transformed into some kind of normalized form. For each command in the normalized form, a hash value is calculated. Matching hash values indicate the same command with only slight differences, such as run-time parameters.

Here is an example of how the text looks before and after processing:

**Before processing:**

SELECT 0 AS Fld1,

 '' AS Fld2,

 0 AS fld3

FROM dbo.v\_dummy

UNION

SELECT dbo.Table1.Fld1 AS Fld1,

 dbo.Table1.Fld2 AS Fld2,

 dbo.Table1.fld3 AS fld3

FROM dbo.Table1, dbo.Table3 AS pers, dbo.Table2

WHERE (dbo.Table1.Fld1 = pers.Fld1)

 AND (dbo.Table1.Fld1 = dbo.Table2.Fld1)

 AND (dbo.Table2.Fld7 = CAST (20 AS INT))

 AND (dbo.Table1.fld4 = 0);

**Normalized:**

SELECT {##} AS FLD1, {STR} AS FLD2, {##} AS FLD3 FROM DBO.V\_DUMMY UNION SELECT DBO.TABLE1.FLD1 FLD1, DBO.TABLE1.FLD2 FLD2, DBO.TABLE1.FLD3 FLD3 FROM DBO.TABLE1, DBO.TABLE3 PERS, DBO.TABLE2 WHERE ( DBO.TABLE1.FLD1 = PERS.FLD1) AND ( DBO.TABLE1.FLD1 = DBO.TABLE2.FLD1 ) AND ( DBO.TABLE2.FLD7 = CAST({##} AS INTEGER) ) AND ( DBO.TABLE1.FLD4 = {##} )

In the database with trace results, the following four tables are most interesting:

* **tblUniqueBatches** – contains the list of unique calls like EXEC with command text in initial and normalized presentation. If EXEC includes different parameters, the HashId for this command will be different.
* **tblUniqueStatements** – contains the list of unique commands within the body of the stored procedure, triggers, and so on. It includes also both initial and normalized text presentations.
* **tblBatches** – contains all individual calls with data on resource consumption for each call.
* **tblStatements** - contains all individual commands with data on resource consumption for each call.

The **tblUniqueBatches** and **tblBatches** tables are related to each other, as well as tables **tblUniqueStatements** and **tblStatements**. You can track their relationships on the HashID, which is the hash value of the query text.

The **tblBatches** and **tblStatements** tables are bound with each other through the command’s execution sequence id– column **BatchSeq**.

This set of tables that contain all the commands and batches set to SQL Server provides you with the perfect environment for comparing commands in different tests.

So, you have two databases: one with results of the traces made on SQL Server 2000, another with results of the traces made on SQL Server 2008.

Each time ReadTrace starts, it cleans up data in the databases. To avoid losing previously collected data, point to a different database every time you run it.

##

## Preparing data for analysis

To start analyzing data, create an empty table with the following structure.

CREATE TABLE tempdb..mins25 (

 mindur BIGINT ,

 maxdur BIGINT ,

 avgdur BIGINT ,

 minreads BIGINT ,

 maxreads BIGINT ,

 avgreads BIGINT ,

 minwrites BIGINT ,

 maxwrites BIGINT ,

 avgwrites BIGINT ,

 mincpu BIGINT ,

 maxcpu BIGINT ,

 avgcpu BIGINT ,

 maxrows BIGINT ,

 testid VARCHAR (30) ,

 normText VARCHAR (300),

 execCount BIGINT ,

 hashid BIGINT

)

Looking at the structure, you can see that these queries collect characteristics of the queries’ execution, their normalized text, and the hash values of the queries.

Let’s query the database’s information for specific test runs to analyze performance.

For SQL Server 2000 traces, use this query.

INSERT INTO tempdb..mins25

SELECT min(duration) AS 'mindur',

 max(duration) AS 'maxdur',

 avg(duration) AS 'avgdur',

 min(reads) AS 'minreads',

 max(reads) AS 'maxreads',

 avg(CAST (reads AS BIGINT)) AS 'avgreads',

 min(writes) AS 'minwrites',

 max(writes) AS 'maxwrites',

 avg(writes) AS 'avgwrites',

 min(cpu) AS 'mincpu',

 max(cpu) AS 'maxcpu',

 avg(cpu) AS 'avgcpu',

 max(rows) AS Maxrows,

 '01sql2000' AS TestID,

 substring(NormText, 1, 300) AS NormText,

 count(\*) AS execCount,

 max(tsu.hashid) AS [HashID]

FROM [sql2000].ReadTrace.tblUniqueStatements AS tsu WITH (NOLOCK)

 INNER JOIN

 [sql2000].ReadTrace.tblStatements AS ts WITH (NOLOCK)

 ON tsu.HashID = ts.HashID

WHERE (normtext LIKE '%Update%'

 OR normtext LIKE '%Insert%'

 OR normtext LIKE '%Select%'

 OR normtext LIKE '%Delete%')

 AND (normtext NOT LIKE '%STATMAN%')

GROUP BY ts.hashid, substring(NormText, 1, 300);

For SQL Server 2005/2008 traces, use this query.

INSERT INTO tempdb..mins25

SELECT min(duration / 1000) AS 'mindur',

 max(duration / 1000) AS 'maxdur',

 avg(duration / 1000) AS 'avgdur',

 min(reads) AS 'minreads',

 max(reads) AS 'maxreads',

 avg(CAST (reads AS BIGINT)) AS 'avgreads',

 min(writes) AS 'minwrites',

 max(writes) AS 'maxwrites',

 avg(writes) AS 'avgwrites',

 min(cpu) AS 'mincpu',

 max(cpu) AS 'maxcpu',

 avg(cpu) AS 'avgcpu',

 max(rows) AS Maxrows,

 'sql2008' AS TestID,

 substring(NormText, 1, 300) AS NormText,

 count(\*) AS execCount,

 max(tsu.hashid) AS [HashID]

FROM [sql2008].ReadTrace.tblUniqueStatements AS tsu WITH (NOLOCK)

 INNER JOIN

 [sql2008].ReadTrace.tblStatements AS ts WITH (NOLOCK)

 ON tsu.HashID = ts.HashID

WHERE (normtext LIKE '%Update%'

 OR normtext LIKE '%Insert%'

 OR normtext LIKE '%Select%'

 OR normtext LIKE '%Delete%')

 AND (normtext NOT LIKE '%STATMAN%')

GROUP BY ts.hashid, substring(NormText, 1, 300);

Did you note that for SQL Server 2008 I used duration / 1000? When SQL Server Profiler writes traces, duration values in SQL Server 2000 are written in milliseconds, whereas in SQL Server 2005 and SQL Server 2008, duration values are written in microseconds. This is why in the second query it's necessary to use (duration/1000) to collect query execution time. Also note that autostats queries (normtext NOT LIKE '%STATMAN%') are excluded from analysis. This is because they could be numerous, and while I would not recommend disabling automatic statistics refresh, they are not necessary for analysis.

In addition to the average values for reads, writes, duration, and processor load, take the maximums, minimums, and number of query executions. Maximum and minimum values help to filter out events that are not relevant. The number of query executions can also be extremely helpful. Consider a query that is executed several hundred or thousand times with a slight increase in execution time. A different query is executed once or twice, but it suffers a much worse performance drop. The first query should be optimized first, because under actual load it has a much higher chance of degrading overall application performance. But the best advice here is to know your app. You know what is important and what is not important, because you know your system better than anybody else; use that knowledge to decide which parameters to track.

##

## Analyzing test result differences

So far you’ve collected all of the necessary information, and you are very close to finding out how the new version of the software will work in your situation. However, the most interesting question remains: Which version performs better? How are you going to decide? This is not an easy task. You can start by looking at the kind of work you need the system to do. For example, consider a highly concurrent OLTP environment. Let’s take a look at what the different metrics in SQL Server Profiler actually tell us in this case:

* **Duration** (query execution time) might not provide necessary information about slow-responding queries. The problem with **Duration** is that it reflects how long the query takes in total, from start to finish, *including* all waits in between. So, if you open two query windows in SQL Server Management Studio and run BEGIN TRAN UPDATE Pubs.Dbo.Authors SET State = 'CA' in one window and the same command in the other window, the second statement will wait until the first commits or rolls back. The duration for the second statement is therefore high, but this does not tell you anything about actual query performance. **Duration** is only useful if queries never have to wait for each other, which would be unusual in a multiuser environment.
* The **CPU** column shows real query execution time, but it doesn’t indicate whether the execution plan was optimal. SQL Server first has to get data pages into the buffer and a significant amount of reads means that it has to bring a large amount of pages into the buffer to process, which may increase CPU time as well.
* The **Writes** column does not provide a good picture of disk activity in this case. Most systems write to disk much less frequently than they read from it, and badly performing plans usually suffer from excessive reads, not writes.
* The number of reads is the criteria I use to check to see whether the plan becomes better or worse. Increasing numbers of reads often means suboptimal optimizer decisions that must be located and improved.

What about a data warehouse or a pure reporting application? This is another story. Without going into much detail, for apps like these, which usually have a much higher volume of queries and many fewer concurrent users, **Duration** may be a good choice.

Now you are ready to query the data you’ve gathered.

SET NOCOUNT ON;

GO

DECLARE @testid AS VARCHAR (30), @hashid AS BIGINT;

DECLARE @testid2008 AS VARCHAR (30);

DECLARE @hashid2008 AS BIGINT;

DECLARE @sql2000reads AS INT;

DECLARE @sql2008reads AS INT;

DECLARE sql2000tests CURSOR

 FOR SELECT avgreads,

 testid,

 hashid

 FROM tempdb..mins25

 WHERE testid = 'Sql2000';

OPEN sql2000tests;

FETCH NEXT FROM sql2000tests INTO @sql2000reads, @testid, @hashid;

WHILE @@FETCH\_STATUS = 0

 BEGIN

 SELECT @sql2008reads = avgreads,

 @testid2008 = testid,

 @hashid2008 = hashid

 FROM tempdb..mins25

 WHERE testid = 'Sql2008'

 AND hashid = @hashid;

 IF @@ROWCOUNT = 0

 BEGIN

 SELECT @sql2008reads = NULL,

 @testid2008 = NULL,

 @hashid2008 = NULL;

 END

 SELECT @sql2000reads AS [SQL 2000 reads],

 @sql2008reads AS [SQL 2008 reads],

 @testid AS [Test ID],

 @hashid AS [Hash ID];

 SELECT @sql2008reads = NULL,

 @testid2008 = NULL,

 @hashid2008 = NULL;

 FETCH NEXT FROM sql2000tests INTO @sql2000reads, @testid, @hashid;

 END

CLOSE sql2000tests;

DEALLOCATE sql2000tests;

I use a cursor to find matches in queries with the same HASHID between test runs on SQL Server 2000 and SQL Server 2008.

This query returns a table with discrepancies between **Reads** columns. You can copy the output into Microsoft Excel® to see the differences between test runs.

It makes a lot of sense to compare not only the absolute values but also the change in percent for all the values. The higher the value in percentage between the test runs, the more attention you should pay to the query. Those with the most significant difference in amount of **Reads** are likely to be more significant for your analysis. Also, look at how often the query was executed.

The following query sorts the table by the value of discrepancy and selects the query texts’ HASHIDs. Some of the more frequent commands might be included in the report many times. In this case, you should sort the table by the level of discrepancy as well as grouping the results by HASHID.

If you have the HASDID for some query, you can find its text to check and improve it if necessary. The following query helps locate a statement text from its HASHID.

SELECT ts.hashid,

 ts.reads,

 tsu.OrigText AS [Statement],

 tbu.OrigText AS [Batch]

FROM sql2008.ReadTrace.tblStatements AS ts

 INNER JOIN

 sql2008.ReadTrace.tblUniqueStatements AS tsu

 ON ts.HashID = tsu.HashID

 INNER JOIN

 sql2008.ReadTrace.tblBatches AS tb

 ON tb.BatchSeq = ts.BatchSeq

 INNER JOIN

 sql2008.ReadTrace.tbluniquebatches AS tbu

 ON tb.HashID = tbu.HashID

WHERE ts.HashID = 1665067467579492691;

The output will contain the query and amount of reads characterizing execution of this query. You may want to sort output by the number of READS. Depending on the parameter values, the same statement may produce different plans and output. Sorting by read activity, you will find the set of parameters that produces the most "heavy" plan. Also you may find that in certain cases SQL Server reuses a plan that is not optimal for this particular set of parameters

# **Testing with database compression implemented**

One of the most interesting new features of SQL Server 2008 is database (table and index) compression. For more information, see [Creating Compressed Tables and Indexes](http://msdn.microsoft.com/en-us/library/cc280449.aspx) (http://msdn.microsoft.com/en-us/library/cc280449.aspx) in SQL Server Books Online. In database compression, SQL Server uses compression algorithms to compress database and index pages, thus reducing memory requirements (more pages fit into memory) and I/O (less data should be taken from disk). Data is compressed both on disk and in memory and uncompressed only when an actual operation is necessary. Compression is a processor-intensive operation, and if your application consumes a lot of processor resources, compression may create unacceptable load. But if you have some processor ticks to spare, try the compression feature in a test environment. You may find that it will be very useful in your environment.

I recommend determining what tables should be compressed (it’s not necessary to compress everything) and what compression level you should use. SQL Server can compress at the row level (which requires less processing power but can sometimes yield a lower compression ratio) or the page level (which starts with row compression and adds additional compression over the whole page). Page compression requires more processor power, but sometimes the additional compression is too small to justify the additional load. SQL Server provides a stored procedure, **sp\_estimate\_data\_compression\_savings**, to collect estimates. You can also use the Data Compression Wizard.



Figure 3: Data Compression Wizard - an easy way to estimate compression

When you choose objects to compress, consider the following:

* The higher the data repetition, the better the compression. Data values such as NULL, 0, and empty strings take no bytes.
* Highly used (ones that are often modified) indexes should not be compressed; if they are compressed, performance may suffer.
* Row compression is often enough, but you should also test page compression to see whether it can provide further improvements (always consider not only storage reduction but also processor load increase).
* These tools provide estimates, not measurements. Always perform tests on a full set of data before you make your decision. (Be sure to use a full set of data: An attempt to use SELECT TOP <N> on an actual table and just compressing that subset of data can also be misleading. In one case, I found a fourfold difference between the top 300,000 and the full set of 10 million records.)

#

# **Possible upgrade issues and workarounds**

Our experience shows that in most standard database systems, the major part of the workload consists of OLTP queries, mixed with a few reporting queries. In these workloads, we have observed that after migration, nothing changes for about 90 percent of the queries. Of the remaining 10 percent, about half ran much more quickly, and about half ran much more slowly.

Here is a chart from one of my actual projects.



Figure 4: Comparison chart.

The x axis shows the query number (which is actually a row number in an Excel spreadsheet). The y axis shows the difference in reads between SQL Server 2000 and SQL Server 2008.

In almost all cases, changed query plans were the main reason for the decrease in query speeds. The migration project where we upgraded from SQL Server 2000 directly to SQL Server 2008 provided many examples of queries whose execution behavior changed after the upgrade. Unfortunately, we found that there was no universal way to identify them and predict those changes, and it was sometimes difficult to find logical a explanation for this changing behavior We were, however, able to figure out how to resolve them: We used index schema changes and hints, and in some cases, we rewrote code.

##

## Index schema changes and hints to the rescue

Migration is dictated not by pure academic interest, but by real business demand and targets to maintain current levels of performance (at a minimum). The best way to deal with problematic queries is to start with SQL Server Database Engine Tuning Advisor. This tool was introduced as Database Tuning Advisor in SQL Server 2000, and SQL Server 2005 and SQL Server 2008 included improvements. Database Engine Tuning Advisor consumes "the load," which can be Transact- SQL scripts or profiler traces, and after analyzing it, suggests indexes and/or statistics that improve query performance.

In this section, we’ll walk through the process of analyzing and repairing a few problems using Database Engine Tuning Advisor.

For example, you may encounter a problem with a query that is part of a stored procedure called **sp\_myMainProcedure**. Our stress load contains an EXEC statement that calls procedure passing it necessary parameters. In my experience, if you pass an EXEC statement to ВЕФ as is, it will refuse to analyze it. So you must expand EXEC into actual procedure code, declaring all necessary parameters as variables.

So instead of this:

EXECUTE sp\_myMainProcedure @parm1=1, @parm2=45, @parm3='Dmitry Artemov', @parm4=12345678', @parm5='20091021'

Use this code:

DECLARE @parm1 AS INT, @parm2 AS INT, @parm3 AS VARCHAR (50), @parm4 AS BIGINT, @parm5 AS DATETIME;

SELECT @parm1 = 1,

 @parm2 = 45,

 @parm3 = 'Dmitry Artemov',

 @parm4 = '12345678',

 @parm5 = '20091021';

-- Remaining procedure code goes here

This variant is acceptable for Database Engine Tuning Advisor, and Database Engine Tuning Advisor can offer some improvements, like creating missing indexes or statistics.

Examine the recommendations in Database Engine Tuning Advisor. First, look at the recommended statistics, because the process of creating statistics is less costly, compared with indexes, and it does not change database schema.

Now you can check to see whether the Database Engine Tuning Advisor suggestions indeed improve performance.

Add the following set of commands before the test script in order to collect execution statistics and the actual plan.

SET STATISTICS IO ON -- shows information related to the I/O of both logical and physical disks

GO

SET STATISTICS PROFILE ON – shows execution plan

GO

SET STATISTICS TIME ON – shows CPU time

GO

First you need to run script **without applying** the recommended changes to collect characteristics of the query execution. Then build statistics and run the script again. If the results are good, you are done. If not, follow the deeper Database Engine Tuning Advisor recommendation and build indexes, and examine how query behavior is affected.

If none of these steps help, you can look into output related to the disk’s I/O. SET STATISTICS IO output might look like this.

Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0

Table 'Table1'. Scan count 808, logical reads 5120, physical reads 0

**Table 'Table2'. Scan count 68430, logical reads 306780,**

Table 'Table3'. Scan count 2, logical reads 980, physical reads 0

Table 'Table4'. Scan count 2, logical reads 2526, physical reads 0

Table 'Table5'. Scan count 2, logical reads 108, physical reads 0

Table 'Table6'. Scan count 2, logical reads 2, physical reads 0

Table 'Table7'. Scan count 2, logical reads 158, physical reads

In this example, the most interesting part is related to Table2, which experienced most of the reads. Let’s analyze partial output of the command SET STATISTICS PROFILE.

EstimateRows Rows Executes

---------------------- ----------------------------------------------------------------

404 404 1 | |--Table Scan(OBJECT:([tempdb].[dbo].[#tempTable]

**771827,1 2995389 1 |--Nested Loops(Inner Join, OUTER REFERENCES:([tnm].[i**

1336,619 34215 1 |--Hash Match(Inner Join, HASH:([tdn].[Field3

20 0 0 | |--Compute Scalar(DEFINE:([Expr1152]=CONVERT

20 20 1 | | |--Clustered Index Scan(OBJECT:([Databa

1976,949 34215 1 | |--Hash Match(Inner Join, HASH:([tsb].[Field

1 0 0 | | |--Table Scan(OBJECT:([t

34215 34215 1 | |--Clustered Index Scan(OBJECT:([Databa

**473,1751 2995389 / 34215 (88) |--Index Seek(OBJECT:([Database].[dbo].[Table2**

Notice the discrepancies in the number of rows between the estimation and the actual output. In **Nested Loops(Inner Join, OUTER REFERENCES:([tnm].[I**, the actual count is about four times higher than the estimate (2995389/771827,1 = 3.9); in **Index Seek(OBJECT:([Database].[dbo].[Table2**, the actual count is about five times higher than the estimate (473.1751/(2995389/34215)=5.4).

This discrepancy comes from the way cost-based optimization works in SQL Server. One of the criteria the optimizer uses is the number of rows are output on each step of the plan. For example, consider a scenario in which two large tables are joined, causing the query to return many rows from each table. In this case the query optimizer may decide on the HASH JOIN, which produces only one scan over the entire table. If the tables in the JOIN differ significantly in size, the optimizer may substitute a LOOP JOIN. If the output amount on certain plan steps is estimated incorrectly, the optimizer creates a suboptimal plan and the server is forced to use more reads than necessary. In this case, the optimizer used bookmarks instead of scanning. If the execution plan produces a very large number of bookmarks for LOOP JOIN, going many times from the nonclustered index to the base table could be much costlier than scanning the table only once with HASH JOIN. It seems that in this example, this is exactly what happened.

For more information about how to read statistics profiles, see [How to Read Statistics Profile](http://blogs.msdn.com/queryoptteam/archive/2006/08/29/730521.aspx) (http://blogs.msdn.com/queryoptteam/archive/2006/08/29/730521.aspx) in the Tips, Tricks, and Advice from the SQL Server Query Optimization Team blog.

In effect, although it returned correct results, the optimizer used a suboptimal plan. Sometimes you can help the optimizer by refreshing statistics (with FULLSCAN if possible) or building new statistics per Database Engine Tuning Advisor recommendations. If it cannot be fixed by updating statistics or building new statistics, you can force the optimizer to use HASH JOIN. This step reduces the amount of READs. Also, HASH JOIN builds hash values of the join fields that require additional CPU cycles. You can also use the hint OPTION (MAXDOP N) to force use of parallel execution plans where possible, to provide more resources for this CPU-intensive operation. (SQL Server Books Online cautions that query hints should be used only as a last resort by experienced developers and administrators.)

In OLTP-class applications, I seldom recommend using a maximum degree of parallelism higher than 1. The default setting of 0 for **max degree of parallelism** can negatively impact performance due to high waits on CXPACKET. However, a reasonable increase in the degree of parallelism may significantly improve a particular query. You can also try to play with different values; try either 2 or 4 and see which value provides better performance. Choose the optimal value, you and then move on to the next step.

Next, take a look at UPDATE. In this example, this very light command slows down on executing selection of the rows to be updated. Again, bookmarks have been used (RID lookup, in this case). Here is a plan that was output by a SET STATISTICS PROFILE command.

Table 'Table1'. Scan count 1, **logical reads 18558**, physical reads 0, read-ahead reads 0,

(9259 row(s) affected)

Rows Executes StmtText

------- --------- ------------------------------------------------------------------------

9259 1 UPDATE [dbo].[Table1] set [Fld1] = [Fld1]+@1 WHERE [Fld2

9259 1 |--Table Update(OBJECT:([Database].[dbo].[Table1]), SET:([Database].[d

0 0 |--Compute Scalar(DEFINE:([Expr1004]=CONVERT\_IMPLICIT(numeric(18,

9259 1 |--Top(ROWCOUNT est 0)

9259 1 |--Nested Loops(Inner Join, OUTER REFERENCES:([Bmk1000]

0 0 |--Compute Scalar(DEFINE:([Expr1016]=BmkToPage([Bm

9259 1 | |--Index Seek(OBJECT:([Database].[dbo].[Table

**9259 9259 |--RID Lookup(OBJECT:([Database].[dbo].[Table1]),**

You may be able to avoid this bottleneck by using a clustered index. After you create the index, use SET STATISTICS PROFILE again.

Table 'Table1'. Scan count 1, **logical reads 41**, physical reads 0, read-ahead reads 0

(9259 row(s) affected)

Rows Executes StmtText

-------------------- -------------------- --------------------------------------------

9259 1 UPDATE [dbo].[Table1] set [Fld1] = [

9259 1 |--Clustered Index Update(OBJECT:([Database

 SQL Server Execution Times:

 CPU time = 780 ms, elapsed time = 777 ms.

The number of reads is significantly lower, but the CPU load is higher.

Here is the CPU time before the index was created.

 CPU time = 260 ms, elapsed time = 304 ms.

After the index is created, the values more than double.

 CPU time = 780 ms, elapsed time = 777 ms.

At the same time, I/O went down from more than 18,000 reads to 41 reads. What is going on? The updated table had dynamic compression enabled on it. This helps to reduce the amount of space the database requires on the hard disk, as well as memory requirements, because data pages are compressed both in memory and on disk. However, this change is reflected in higher CPU, because updating data pages will require decompressing data in the memory buffer, which adds additional CPU time. When only tables were compressed, using an index seek didn’t cause an increase in CPU load, and updating was short and cheap. As soon a created clustered index was created (which means there is now a sorted table) requests like searching the value or updating are reflected in additional CPU cost.

If you decompress the table and leave clustered index, the CPU time decreases dramatically.

logical reads 102

CPU time = 70 ms, elapsed time = 74 ms

This is almost an ideal picture. The number of reads increases only slightly, and the CPU cost is very low. But now there is no data compression, which is troublesome: the table is very big. Is it possible to update rapidly without sacrificing additional disk space for data storage? Table partitioning may help achieve this. If the data modification touches only a small range of data (such as the freshest data), partitioning might help by applying compression on data and indexes for some data ranges that are not frequently queried.

Next we addressed a particularly troublesome stored procedure. In SQL Server 2008, it worked badly compared with SQL Server 2000. We found that when we ran it as "EXECUTE ProcName @param1, @param2" it took 9 minutes to finish, but when we copied its full text into the query window and declared all necessary parameters, it took only 30 seconds to finish, even though the code was exactly the same. This is due to a phenomenon called *parameter sniffing*. If you run code as a big batch and declare the parameters, SQL Server builds a plan knowing the values of the parameters. If you do not declare the parameters, the optimizer uses parameters from a previously created plan. This can result in the creation of a suboptimal plan. After profiling code execution, I found two statements that were responsible for bad performance. I first tried to force recompile on the statements using OPTION (RECOMPILE). It helped, but only for the second statement. For the first, I called Database Engine Tuning Advisor for help and implemented the recommended index. This resulted in an execution time of 27 seconds.

##

## Code rewrite

When you upgrade to latest newer version of SQL Server, it is good idea to set aside some time to rethink some coding approaches. I do not recommend addressing application architecture limitations. The more changes you implement, the higher risk. But code review is highly recommended. First of all, try to address old style code that looks like this.

SELECT t1.f1, t1.f2, t2.f3, t2,f5

FROM table1 t1, table2 t2

WHERE t1.f1=t2.f1

The statement above could be changed to the following:

SELECT t1.f1, t1.f2, t2.f3, t2,f5 FROM table1 t1

JOIN

table2 t2

ON t1.f1=t2.f1

The second version enables hints to be implemented quickly if necessary.

SELECT t1.f1, t1.f2, t2.f3, t2,f5 FROM table1 t1

**INNER HASH JOIN**

table2 t2

ON t1.f1=t2.f1

On several occasions I also found that for complex queries (such as those that include five or six joins), rewriting from WHERE t1.f1=t2.f1 to JOIN table2 t2 ON t1.f1=t2.f1 significantly improved the execution plan and performance.

Check critical pieces of code. The optimizer may not be able to create a good plan if the criteria are too complex.

For example, in the following query, the OR operator limits the range of options for the optimizer.

SELECT MAX(l.fld1)

FROM dbo.Table1 l (NOLOCK)

WHERE ( l.fld2 = @parm1 OR l.fld3 = @parm1 ) AND ( l.fld4 <= @psrm2 )

Therefore, the reads and CPU time are very high.

logical reads 2010

CPU time = 2010 ms

However, if you split the query so that it chooses the maximum of two possible variables, performance improves.

DECLARE @max1 AS DATETIME, @max2 AS DATETIME, @trueMax AS DATETIME;

SET @max1 = (SELECT MAX(l.fld3)

 FROM dbo.Table1 AS l WITH (NOLOCK)

 WHERE (l.Fld1 = @parm1)

 AND (l.fld2 <= @parm2));

SET @max2 = (SELECT MAX(l.fld3)

 FROM dbo.table1 AS l WITH (NOLOCK)

 WHERE (l.fld4 = @parm3)

 AND (l.fld2 <= @parm2));

SET @trueMax = (SELECT CASE WHEN @max1 < @max2 THEN @max2 ELSE @max1 END);

If a proper index is in place, the following is returned. Reads are 670 times better and CPU 2,000 times better.

-- Calculate max1

logical reads 3

 CPU time = 0 ms

-- Calculate max2

logical reads 3

 CPU time = 0 ms

This is a very good outcome, especially if the query is run frequently.

## Rewriting code when code is not available

As I said, rewriting code is sometimes necessary. But what if the procedure source is not available to you, or what if you are bound by legal obligations? What if the problem is in the middle-tier component that sends Transact-SQL to the server?

SQL Server 2008 introduces plan guides, a feature that enables you to change code without touching it. For more information about plan guides, see [Optimizing Queries in Deployed Applications by Using Plan Guides](http://technet.microsoft.com/en-us/library/ms187032.aspx) (http://technet.microsoft.com/en-us/library/ms187032.aspx) and [Query Hints](http://msdn.microsoft.com/en-us/library/ms181714.aspx) (http://msdn.microsoft.com/en-us/library/ms181714.aspx) in SQL Server Books Online. SQL Server Books Online says that "Plan guides can be useful when a small subset of queries in a database application deployed from a third-party vendor are not performing as expected. Plan guides influence optimization of queries by attaching query hints or a fixed query plan to them."

For more information about working with query plans, including the USE PLAN query hint and plan guides, see the white paper [Forcing Query Plans](http://technet.microsoft.com/en-us/library/cc917694.aspx) (http://technet.microsoft.com/en-us/library/cc917694.aspx).

# **Conclusion**

Upgrading to SQL Server 2008 provides administrators and developers with a rich set of programming and diagnostic tools. Careful planning, sufficient testing, and a proper test environment can help you avoid many issues that may arise during and after an upgrade. You can use the tools and methodologies presented in this white paper to perform upgrades as well as for future development.

# **References**

The sites in this list contain a wealth of information that you can use for further optimization.

**Storage optimization.**

Disk Partition Alignment Best Practices for SQL Server (<http://msdn.microsoft.com/en-us/library/dd758814.aspx>)

Predeployment I/O Best Practices (<http://www.microsoft.com/technet/prodtechnol/sql/bestpractice/pdpliobp.mspx>)

Microsoft SQL Server I/O Basics Chapter 2 (http://technet.microsoft.com/en-us/library/cc917726.aspx)

**Storage blog**

Paul Randall blog (<http://sqlskills.com/blogs/paul/> )

**Upgrade references**

Resources for Upgrading to SQL Server 2008 (<http://technet.microsoft.com/en-us/library/cc936623.aspx>)

SQL Server 2005 Upgrade Technical Reference Guide (http://www.microsoft.com/downloads/details.aspx?FamilyID=3d5e96d9-0074-46c4-bd4f-c3eb2abf4b66&displaylang=en)

SQL Server 2008 Upgrade Technical Reference Guide (<http://www.microsoft.com/downloads/details.aspx?FamilyID=66d3e6f5-6902-4fdd-af75-9975aea5bea7>)

**Performance references**

Plan Caching in SQL Server 2008 (<http://technet.microsoft.com/en-us/library/ee343986.aspx>)

Statistics Used by the Query Optimizer in Microsoft SQL Server 2008 (<http://technet.microsoft.com/en-us/library/dd535534.aspx>)

Troubleshooting Performance Problems in SQL Server 2008 (<http://technet.microsoft.com/en-us/library/dd672789.aspx>)

**For more information**

SQL Server Web site (<http://www.microsoft.com/sqlserver>)

SQL Server TechCenter (<http://technet.microsoft.com/en-us/sqlserver>)

SQL Server DevCenter (<http://msdn.microsoft.com/en-us/sqlserver>)

SQL Server Customer Advisory Team Web site (<http://sqlcat.com/>)

**Tools mentioned in this white paper**

* RML Utilities for SQL Server (x86) – <http://www.microsoft.com/downloads/details.aspx?FamilyId=7EDFA95A-A32F-440F-A3A8-5160C8DBE926&displaylang=en>
* RML Utilities for SQL Server (x64) – <http://www.microsoft.com/downloads/details.aspx?FamilyId=B60CDFA3-732E-4347-9C06-2D1F1F84C342&displaylang=en>
* Text Workbench (trial version) [http://www.silveragesoftware.com/hffr.htmlhttp://www.silveragesoftware.com/hffr.html](http://www.silveragesoftware.com/hffr.html)
* XML Notepad 2007 <http://www.microsoft.com/downloads/details.aspx?familyid=72d6aa49-787d-4118-ba5f-4f30fe913628&displaylang=en>
* Microsoft SQL Server 2008 Upgrade Advisor - http://www.microsoft.com/downloads/details.aspx?familyid=F5A6C5E9-4CD9-4E42-A21C-7291E7F0F852&displaylang=en