

# **“The SQL Server and .NET Blog” eBook Series**

## **Tuning SQL Server**

**Artemakis Artemiou**

**“The SQL Server and .NET Blog”  
eBook Series**

**Tuning SQL Server**

ARTEMAKIS ARTEMIOU

PUBLISHED BY  
Artemakis Artemiou

Copyright © 2013 Artemakis Artemiou

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Microsoft and the trademarks listed at <http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx> are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

The example companies, organizations, products, domain names, email addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the author, nor distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

## About the Author

**Artemakis Artemiou** is a Senior SQL Server Architect and a [SQL Server MVP](#). He holds many certifications (MCTS, MCITP) on different versions of SQL Server. Artemakis is the president of the Cyprus .NET User Group ([CDNUG](#)) and the [INETA-Europe](#) Country Leader for Cyprus. He regularly writes articles on different topics on SQL Server and publishes them on his blog or on other blogs/websites as guest articles. Artemakis participates as a speaker on many local user group events as well as an invited speaker on local Microsoft conferences and workshops. You can find his blog at: <http://aartemiou.blogspot.com>. You can also find him on Twitter at: <http://twitter.com/artemakis>

*To all of those who never stop seeking knowledge,  
who never stop sharing knowledge without  
expecting anything in return, for those  
who just want to help their fellow man.*  
-A.A.

# Table of Contents

	<b>Introduction</b>	8
<b>Chapter 1</b>	<b>Indexes</b>	11
	▪ Getting Table Index Information in SQL Server	12
	▪ Index Fragmentation in SQL Server (Reorganizing/Rebuilding Indexes)	12
	▪ How to Rebuild all the Indexes of a Database in SQL Server	17
	▪ Summary	19
<b>Chapter 2</b>	<b>Log Files Management</b>	20
	▪ Retrieving Log Space Information within a SQL Server Instance	21
	▪ Retrieving Log Space Information within a SQL Server Instance - The Stored Procedure!	22
	▪ Handling Disk Space Issues During Heavy Index Rebuild Operations	23
	▪ Executing Heavy Set-Based Operations Against VLDBs in SQL Server	25
	▪ Summary	27
<b>Chapter 3</b>	<b>Locking and Blocking</b>	28
	▪ Monitoring Locking in SQL Server	29
	▪ Updating SQL Server Tables Using Hints	30
	▪ Table-Level Locking Hints in SQL Server	31
	▪ Summary	33
<b>Chapter 4</b>	<b>TempDB</b>	34
	▪ TempDB Growth	35
	▪ Where are Temporary Tables Stored in SQL Server?	36
	▪ Summary	37
	<b>List of Listings</b>	39
	<b>List of Figures</b>	40



# Introduction

I started working with SQL Server and .NET (C#) more than ten years ago. Since then it has been quite a journey! Each release came - and still comes - with exciting new features enabling us to do more and more! Every time waiting for that new built, in order to start testing it, exploring it, learning it, as soon as it becomes available. The possibilities are endless! The only limit is your creativity!

The massive interaction with the SQL Server community started at about seven years ago. Blogging, organizing user group events, speaking in user group meetings, conferences and other events, open-source projects related to SQL Server, guest articles, discussions on message boards/forums, and much more!

The love for technical writing and knowledge sharing urged me for adding another activity to my interaction with the community that is **book authoring**. In order to be able to write, you first need to acquire and comprehend the specific technical knowledge. You need to explore, to experiment, to test. You need to test the limits of each new technology or feature in order to be able to fully understand its nature and capabilities.

SQL Server is a powerful data platform that includes several data management and analysis technologies that allow you to do just about anything. Since 2002 I have been exploring SQL Server in many areas. I have been deep diving into various topics of SQL server having to do mainly with: **administration**, **development/data access** and **performance tuning**. These are the three areas of SQL Server I like the most and on which I base, but not limit, my interaction with SQL Server and acquisition of knowledge. To this end, I have decided to publish three eBooks on the three above mentioned areas of SQL Server.

The content of the eBooks will be mainly based on articles that I have already published on my blog. It will actually be a collection of selected articles I have already published, edited and enriched with additional content. This is the reason the series is called “The SQL Server and .NET Blog” eBook Series. This eBook is the first one and it is dedicated to SQL Server performance tuning. Last but not least, note that under the title of each chapter, there will be a URL pointing to the original article on my blog. By following the link, you will be able to leave comments for the articles in this eBook.

## Who Should Read This Book?

---

This book is for database administrators and architects who monitor and tune SQL Server instances in order to keep them operating to the maximum possible performance and stability. The book suggests several techniques that can be used for ensuring a performant SQL Server instance. However, the book is not intended to be a step-by-step comprehensive guide. Additionally, it assumes at least intermediate-level experience with SQL Server administration and knowledge of basic database principles (i.e. indexing, locking, etc.).



## Supported SQL Server Versions

---

Each article in this book has in the end an “**Applies to:**” note indicating the SQL Server versions against which the provided T-SQL scripts can be executed. There are also articles providing different T-SQL scripts for SQL Server 2000, SQL Server 2005 or later. Every time I write an article, I try to write code that can be executed in all versions of SQL Server unless stated otherwise. Even though SQL Server 2000 is not officially supported anymore, I am quite sure that still there are many of you out there who use this version of SQL Server for various reasons (i.e. application compatibility issues, etc.). My advice is to consider upgrading to a newer version of SQL Server the soonest possible, in order to be able to use the benefits of the latest SQL Server releases as well as being able to get official support and updates.

## How Is This Book Organized?

---

This book is organized as follows. Chapter 1 discusses the usage of indexes and the derived benefits of using them along with techniques to maintain them healthy in order to ensure performance. Chapter 2 discusses ways of managing log files as well as how to ensure that their growth will not create issues during the execution of heavy set-based operations. Chapter 3 discusses how you can monitor locking and tracking blocking cases in SQL Server as well as other locking-related topics. Chapter 4 discusses topics having to do with maintaining the good health and performance of the tempdb database as well as retrieving information from tempdb’s data structures.

## Upcoming Books of this Series

---

This is the first eBook of the series “*The SQL Server and .NET Blog*”. There are still two eBooks to be published. The second book will be dedicated to SQL Server administration and will be titled “*Administering SQL Server*”. The third and last book will be dedicated to SQL Server development and data access and will be titled “*Developing SQL Server*”.

## Feedback

---

I am a proud member of the worldwide SQL Server community. Feedback from you, my fellow community members, is more than welcome. Please feel free to visit the following link and provide your feedback:

<http://sqlbooks.aartemiou.com/feedback>

The survey is short. It will only take 5 minutes of your valuable time.

## Acknowledgments

---

Writing a book is not an easy thing. It doesn't really matter the length of the book. Even if you just write a few pages, it takes a considerable amount of time and energy. In the case where you are not a professional technical writer but just a technical community guy like me, this amount of time is valuable "free" time after work, time from your family and beloved ones. You may use this time because you are just passionate in what you do. However, this is not enough, at least to me. Without the support of your family it just doesn't feel right. During this process I had all the support I needed. I am grateful for all the support my family gave me during writing this book and for all their support and love in everything I do. Without their support and encouragement none of this would have been possible.

- Artemakis

## Stay in Touch

---

Feel free to connect! You can find me on the following online channels:

- The SQL Server and .NET Blog: <http://aartemiou.blogspot.com>
- The SQL Server and .NET TV: <http://www.youtube.com/user/sqlserverdotnetblog>
- Personal Website: <http://www.aartemiou.com>
- Twitter: <http://twitter.com/artemakis>
- SQLBooks: <http://sqlbooks.aartemiou.com>

## CHAPTER 1

# Indexes

### **IN THIS CHAPTER:**

- Getting Table Index Information in SQL Server
- Index Fragmentation in SQL Server (Reorganizing / Rebuilding Indexes)
- How to Rebuild all the Indexes of a Database in SQL Server

**Imagine** that you have a large book and you are looking for a specific piece of information. If the book has 500 pages and has no index you will have to go through page by page until to find the information you are looking for. The worst case scenario would be to check all 500 pages. That would be a full “scan” of the book. However, if the book has an index, you will just go through the index list and find the page that contains the information you are looking for.

The same thing happens with databases. In the database world, seeking specific information in a frequently-used large table without using an index, could end-up scanning the entire table thus taking a considerable amount of time. However, when having an index, eventually the data retrieval time would be much faster as the operation will not have to search every row in the database table every time the database table is accessed.

Even though indexes are key data structures in databases, they bring the cost of additional writes and the use of more storage space to maintain the extra copy of data. In addition, they need to be frequently maintained in order to be as performant as possible. Nevertheless, the performance gain when using indexes is significant thus making the overheads of maintaining them negligible.

In order to have a healthy set of indexes you need to run frequent maintenance operations. Such operations include index fragmentation checks, reorganization and rebuild actions.

This chapter includes selected articles from my blog having to do with maintaining indexes in order to ensure performance.

## ■ Getting Table Index Information in SQL Server

(source: <http://aartemiou.blogspot.com/2009/01/getting-table-index-information-in-sql.html>)

In [SQL Server](#), there is a system stored procedure called [sp\\_helpindex](#). This stored procedure is a fast way of getting index information for specific tables.

The syntax for using it is the following:

```
EXEC sp_helpindex 'schema.table_name';  
GO
```

**Listing 1.1:** Retrieving index information using sp\_helpindex.

The execution of the stored procedure will return the following information for all the indexes in the given table:

- index name
- index description (clustered, nonclustered, etc.)
- index keys

➔ *Applies to: SQL Server 2000 or later.*

## ■ Index Fragmentation in SQL Server (Reorganizing/Rebuilding Indexes)

(source: <http://aartemiou.blogspot.com/2009/04/index-fragmentation-in-sql-server.html>)

### ■ Overview

SQL Server automatically maintains indexes whenever insert, delete, or update operations are performed on the underlying data.

However, in cases where the underlying data is actually a large amount of information, the information stored in the indexes becomes scattered over time. This is known as **Index Fragmentation**. Index fragmentation can degrade performance so the indexes must be properly maintained on a regular basis.

There are two types of Index Fragmentation: (i) *External*, and (ii) *Internal*. External fragmentation is when the logical order of the pages in an index does not match the physical order. Internal fragmentation is when the index pages are not filled to the current [fill factor](#) level.

Either way, when you are dealing with index fragmentation (internal, external, or both) it means that you will most probably experience performance degradation on the database which uses those indexes.

To this end, you need to frequently check the fragmentation percentage of the indexes and take the necessary actions whenever is needed. These actions include **rebuilding** or **reorganizing** the fragmented indexes.

- **Collecting Index Fragmentation Statistics**

So let's take one thing at a time. First of all you need to find the fragmentation percentage for the indexes in a database or table.

There are two ways of doing that: (i) by using the '**DBCC SHOWCONTIG**' command, and (ii) by using the '**sys.dm\_db\_index\_physical\_stats**' Dynamic Management View (DMV). The former is supported in all current versions of SQL Server, while the latter is supported in SQL Server 2005 and later.

The DBCC SHOWCONTIG command provides important information about indexes. To this end you can get information like: Object Name, Object ID, Index Name, Number of Pages, Extends, Logical Fragmentation, Extended Fragmentation, etc.

Though, the sys.dm\_db\_index\_physical\_stats DMV provides even more information and it is more robust. To this end, you can get index statistics like: database\_id, object\_id, index\_id, partition\_number, index\_type\_desc, index\_depth, avg\_fragmentation\_in\_percent, fragment\_count, avg\_fragment\_size\_in\_pages, page\_count, avg\_page\_space\_used\_in\_percent, record\_count, compressed\_page\_count, etc.

**\*\*\* Actually, the sys.dm\_db\_index\_physical\_stats DMV replaces DBCC SHOWCONTIG because the latter is deprecated and it will be removed in a future version of SQL Server.**

The above DMV takes five input parameter specifying the following: the database id, object id, index id, partition mode (if you want to get information for a specific partition of an object), and the mode which specifies the scan level during the statistics collection process.

- **DBCC SHOWCONTIG Usage (SQL Server 2000)**

**Syntax Examples:**

```
-- OPTION 1 - Getting Index Fragmentation Information for a
-- single table
USE [DATABASE_NAME];
GO
DBCC SHOWCONTIG ('TABLE_NAME') WITH ALL_LEVELS, TABLERESULTS,
NO_INFOMSGS;
GO

-- OPTION 2 - Getting Index Fragmentation Information for an
-- entire database
USE [DATABASE_NAME];
GO
DBCC SHOWCONTIG WITH ALL_LEVELS, TABLERESULTS, NO_INFOMSGS;
GO
```

**Listing 1.2:** Retrieving index fragmentation in SQL Server 2000.

**Index Fragmentation Percentage in 'DBCC SHOWCONTIG' results:** The important information here is the **LogicalFragmentation** column. As the column's name explains, it displays the fragmentation percentage for the specific index.

- **sys.dm\_db\_index\_physical\_stats Usage (SQL Server 2005 or later)**

**Syntax Examples:**

```
-- OPTION 1 - Getting Index Fragmentation Information for a specific index
SELECT *
FROM sys.dm_db_index_physical_stats(Db_id('DATABASE_NAME'),
Object_id('DATABASE.SCHEMA.TABLE_NAME'),INDEX_ID,NULL,NULL);
GO

-- OPTION 2 - Getting Index Fragmentation Information for a single table
SELECT *
FROM sys.dm_db_index_physical_stats(Db_id('DATABASE_NAME'),
Object_id('DATABASE.SCHEMA.TABLE_NAME'),NULL,NULL,NULL);
GO

-- OPTION 3 - Getting Index Fragmentation Information for an entire database
SELECT *
FROM sys.dm_db_index_physical_stats(Db_id('DATABASE_NAME'),NULL,NULL,NULL,NULL);
GO
```

**Listing 1.3:** Retrieving index fragmentation in SQL Server 2005 or later.

**Index Fragmentation Percentage in 'sys.dm\_db\_index\_physical\_stats' results:**

The important information here is the **avg\_fragmentation\_in\_percent** column. Just like the LogicalFragmentation column of DBCC SHOWCONTIG, it displays the fragmentation percentage for the specific index.

- **Evaluating and utilizing Index Fragmentation Statistics**

With the above two methods, we saw how we can get various index statistics and the most important of them (in the context of this post); the Logical Index Fragmentation.

The question now is what do we do with this information, how can it help for deciding how to *defragment* the necessary indexes and restore the degraded performance of the database?

The question actually is: “When do you decide that you need to reorganize an index, and when to rebuild it?”

As the following [MSDN Library article](#) suggests, whenever the avg\_fragmentation\_in\_percent is **between 5-30%** then you need to **reorganize** the index. When the avg\_fragmentation\_in\_percent is **greater than 30%**, then you need to **rebuild** the index.

**Pseudocode:**

```

Get index fragmentation statistics for database/table/index
IF Logical Fragmentation >=5 AND Logical Fragmentation <= 30 THEN
    REORGANIZE INDEX(es)
ELSE IF Logical Fragmentation >30 THEN
    REBUILD INDEX(es)
END IF

```

**Listing 1.4:** Pseudocode for assessing index fragmentation.

- **Rebuilding an Index**

Rebuilding an index is an efficient way to reduce fragmentation. It automatically drops and re-creates the index thus removing fragmentation. It also reclaims disk space and reorders the index rows in contiguous pages. This process can run online or offline. Note that online index operations are available only in SQL Server Enterprise, Datacenter, Developer, and Evaluation editions.

**Syntax examples for rebuilding indexes in SQL Server 2005 or later:**

```

-- Rebuild a specific index with using parameters
USE [DATABASE_NAME];
GO
ALTER INDEX [INDEX_NAME] ON [SCHEMA.TABLE]
REBUILD WITH (FILLFACTOR=[FILL_FACTOR_VALUE_BETWEEN_0_100], ONLINE=[ON|OFF]);
GO

-- Rebuild all indexes in a table with using parameters
USE [DATABASE_NAME];
GO
ALTER INDEX ALL ON [SCHEMA.TABLE]
REBUILD WITH (FILLFACTOR=[FILL_FACTOR_VALUE_BETWEEN_0_100], ONLINE=[ON|OFF]);
GO

```

**Listing 1.5:** Rebuilding indexes in SQL Server 2005 or later.

\* If you do not want to use parameters, then just remove the 'WITH ...' part.

\* In SQL Server 2000 you cannot use the [ALTER INDEX](#) but you can use the [DBCC DBREINDEX](#) statement instead.

**Syntax examples for rebuilding indexes in SQL Server 2000:**

```

-- Rebuild a specific index in a given table
USE [DATABASE_NAME];
GO
DBCC DBREINDEX ([TABLE_NAME], '[INDEX_NAME]', [FILL_FACTOR_VALUE_BETWEEN_0_100]);
GO

-- Rebuild all the indexes in a given table
USE [DATABASE_NAME];
GO
DBCC DBREINDEX ([TABLE_NAME], '', [FILL_FACTOR_VALUE_BETWEEN_0_100]);
GO

```

**Listing 1.6:** Rebuilding indexes in SQL Server 2000.

- **Reorganizing an Index**

Reorganizing an index physically reorders the leaf-level pages to match the logical, left to right, order of the leaf nodes. It also compacts the index pages based on the existing fill factor value.

**Syntax examples for reorganizing indexes in SQL Server 2005 or later:**

```
-- Reorganize a specific index in a given table
USE [DATABASE_NAME];
GO
ALTER INDEX [INDEX_NAME] ON [SCHEMA.TABLE_NAME]
REORGANIZE;
GO

-- Reorganize all indexes in a given table
USE [DATABASE_NAME];
GO
ALTER INDEX ALL ON [SCHEMA.TABLE_NAME]
REORGANIZE;
GO
```

**Listing 1.7:** Reorganizing indexes in SQL Server 2005 or later.

\* In SQL Server 2000 you cannot use the [ALTER INDEX](#) but you can use the [DBCC INDEXDEFRAG](#) statement instead.

**Syntax example for reorganizing indexes in SQL Server 2000:**

```
DBCC INDEXDEFRAG ([DATABASE_NAME], [TABLE_NAME], [INDEX_NAME]);
GO
```

**Listing 1.8:** Reorganizing indexes in SQL Server 2000.

- **Remarks and Conclusions**

Even though DBCC SHOWCONTIG, DBCC DBREINDEX, and DBCC INDEXDEFRAG are still supported in the latest versions of SQL Server (2005, 2008/R2, 2012), they will be removed in a future version of SQL Server as they are replaced by sys.dm\_db\_index\_physical\_stats and ALTER INDEX respectively. To this end, it is advised to avoid using those features in new development work. Also you will need to have this in mind with respect to any database applications you may maintain.

The reorganization of indexes **always** runs online, while the rebuild provides you with the option to run online or offline. Though, note that in order to be able to rebuild indexes online you must use SQL Server Enterprise, Datacenter, Developer, and Evaluation editions.

Index defragmentation is very important, especially in large databases where insert, delete or update operations are frequently performed. Usually DBAs include automated defragmentation processes in SQL Server maintenance plans in order to prevent performance degradation due to index fragmentation.



## ■ How to Rebuild all the Indexes of a Database in SQL Server

(source: <http://aartemiou.blogspot.com/2009/06/how-to-rebuild-all-indexes-of-database.html>)

In the previous article we talked about Index Fragmentation in SQL Server in terms of how to track it, evaluate it and resolve it. The techniques explained in the article provided ways of reorganizing/rebuilding specific or all the indexes within a given table. But what about when the DBA needs to rebuild all the indexes within a database? How can he achieve this?

It is a fact that in some cases where a large number of indexes in a database have a large percentage of fragmentation, then the recommended approach is to rebuild those indexes. To this end, the worst case scenario would be the DBA to need to rebuild the indexes in all the tables of the database.

Under normal circumstances there is not a direct way allowing rebuilding all the indexes of a database with a single command. A workaround is to run different rebuild statements for each table. Though, I know that workarounds are not very desirable in many cases as they might demand a large amount of time. Luckily, in SQL Server there is the undocumented stored procedure **sp\_MSforeachtable** which allows for recursively executing a T-SQL statement (or more) for all the tables within a database with the use of a single line of code.

Hereunder I propose the syntax on how to rebuild all the indexes of a given database by utilizing the **sp\_MSforeachtable** stored procedure.

### SQL Server 2000

```
--Rebuild all indexes with keeping the default fill factor for each index
USE [DATABASE_NAME];
GO
EXEC sp_MSforeachtable @command1="print '?' DBCC DBREINDEX ('?')";
GO

--Rebuild all indexes with specifying the fill factor
USE [DATABASE_NAME];
GO
EXEC sp_MSforeachtable @command1="print '?' DBCC DBREINDEX ('?', ' ',
[FILL_FACTOR_PERC])";
GO
```

**Listing 1.9:** Rebuilding all indexes for a database in SQL Server 2000.

**SQL Server 2005/2008**

You can either use the syntax provided above for SQL Server 2000 or this one:

```
--Rebuild all indexes online with keeping the default fill factor for each index
USE [DATABASE_NAME];
GO
EXEC sp_MSforeachtable @command1="print '?'", @command2="ALTER INDEX ALL ON ?
REBUILD WITH (ONLINE=ON)";
GO

--Rebuild all indexes offline with keeping the default fill factor for each index
USE [DATABASE_NAME];
GO
EXEC sp_MSforeachtable @command1="print '?'", @command2="ALTER INDEX ALL ON ?
REBUILD WITH (ONLINE=OFF)";
GO

--Rebuild all indexes online with specifying the fill factor
USE [DATABASE_NAME];
GO
EXEC sp_MSforeachtable @command1="print '?'", @command2="ALTER INDEX ALL ON ?
REBUILD WITH (FILLFACTOR=[FILL_FACTOR_PERC],ONLINE=ON)";
GO

--Rebuild all indexes offline with specifying the fill factor
USE [DATABASE_NAME];
GO
EXEC sp_MSforeachtable @command1="print '?'", @command2="ALTER INDEX ALL ON ?
REBUILD WITH (FILLFACTOR=[FILL_FACTOR_PERC],ONLINE=OFF)";
GO
```

**Listing 1.10:** Rebuilding all indexes for a database in SQL Server 2005 or later.

**Considerations:**

- [DBCC DBREINDEX](#) is always an **offline** operation.
- Up to SQL Server 2008 R2, [online index rebuild](#) fails for the following cases:
  - XML index
  - Spatial index
  - Large object data type columns: image, text, ntext, varchar(max), nvarchar(max), varbinary(max), and xml
- SQL Server 2012 or later [supports](#) online index rebuild for the following data types:
  - varchar(max)
  - nvarchar(max)
  - varbinary(max)
  - XML data

## ■ Summary

In this chapter, you learned what indexes are in the database world, their significance, and ways to retrieve information about them, as well as ways to maintain them in order to ensure performance. The next chapter talks about managing log files in SQL Server. Managing log files is another important aspect of SQL Server Database Engine as it has to do not only with performance but with stability as well.

## CHAPTER 2

# Log Files Management

### IN THIS CHAPTER:

- Retrieving Log Space Information within a SQL Server Instance
- Retrieving Log Space Information within a SQL Server Instance - The Stored Procedure!
- Handling Disk Space Issues During Heavy Index Rebuild Operations
- Executing Heavy Set-Based Operations Against VLDBs in SQL Server

**SQL Server** can host huge databases serving as a modern data platform for a wide range of systems and purposes. However, when it comes to running complex operations against very large databases (VLDBs) you need to ensure that different factors which may affect performance are taken into consideration (i.e. log files growth, database size auto-increment, etc.) having to do with the operation's execution.

One significant factor when it comes to executing heavy set-based operations against VLDBs in SQL Server is the growth of the log file(s). If "Autogrowth" is enabled then the log file will keep on getting larger until the set-based operation is completed. In the case where the disk onto where the log file is stored runs out of free space, then the operation's execution will fail. To this end, prior to running such operations you first need to take into consideration scenarios like the above and apply actions in order to minimize the possibilities of having the operation fail.

This chapter discusses ways of managing log files as well as how to ensure that their growth won't create issues during the execution of heavy set-based operations.

## ■ Retrieving Log Space Information within a SQL Server Instance

(source: <http://aartemiou.blogspot.com/2012/03/retrieving-log-space-information-within.html>)

In the everyday life of a Database Administrator there is the task of maintaining the database logs in terms of size they occupy on the disk. Of course, there are automated maintenance and reporting procedures for this task as well as for many other tasks but it is not few the cases where the DBA needs to manually maintain a SQL Server instance or at least some aspects of it! The fastest way to get log space information for all the databases under a SQL Server instance is to use the following [T-SQL command](#):

```
DBCC SQLPERF(LOGSPACE);
GO
```

**Listing 2.1:** Retrieving log space information.

The above command returns a record for each database under the current SQL Server instance which contains the following information:

- **Database Name**
- **Log Size (MB)**
- **log Space Used (%)**
- **Status**

Additionally, if you like to process this information, you can do so by first storing it into a temporary table. You can do this as follows:

```
--Step 1: If temporary table exists, then drop it
IF Object_id(N'tempdb..#tempTbl') IS NOT NULL
DROP TABLE #temptbl;
GO

--Step 2: Create temporary table
CREATE TABLE #temptbl
( dbname VARCHAR(250),
  logsize FLOAT,
  logspaceused FLOAT,
  [status] INT
);

--Step 3: Populate temporary table with log size information
INSERT INTO #tempTbl
EXEC('DBCC SQLPERF(LOGSPACE)');

--Step 4: Process the temporary table
--Examples:
SELECT *
FROM #tempTbl
ORDER BY logsize DESC;
GO
```

**Listing 2.2:** Storing log space information in a temporary table.

➔ **Applies to: SQL Server 2000 or later.**

## ■ Retrieving Log Space Information within a SQL Server Instance - The Stored Procedure!

(source: [http://aartemiou.blogspot.com/2012/03/retrieving-log-space-information-within\\_15.html](http://aartemiou.blogspot.com/2012/03/retrieving-log-space-information-within_15.html))

In the above article, we saw how we can retrieve log space information for all the databases within a SQL Server instance using the command **DBCC sqlperf(logspace)**.

In this article we will refine the process even more by creating a stored procedure that returns log space information for a given database. Here's how:

```
-- CREATING THE STORED PROCEDURE
--Select the database in which the stored procedure will be created
USE [spDBName];
GO

--Check if the stored procedure exists and if so drop it
IF OBJECT_ID('spDBLogInfo') IS NOT NULL
    DROP PROCEDURE spDBLogInfo;
GO

--Create the stored procedure.
--The only input parameter will be the database name.
CREATE PROCEDURE spDBLogInfo
@DBname NVARCHAR (250)
AS
BEGIN
    SET nocount ON;

    -- Main logic
    --Step 1: Create temporary table
    CREATE TABLE #temptbl
    (
        DBName NVARCHAR(250),
        logsize FLOAT,
        logspaceused FLOAT,
        [status] INT
    );

    --Step 2: Populate temporary table with log size information
    INSERT INTO #temptbl
    EXEC('DBCC sqlperf(logspace)');

    --Step 3: Process the temporary table
    SELECT DBName,
        ROUND(logsize,2) as 'Log Size (MB)',
        ROUND(logspaceused,2) as 'Log Space Used (%)',
        ROUND((logsize-(logsize*(logspaceused/100))),2) as 'Available Log Space (MB)'
    FROM #temptbl
    WHERE dbname=@DBname;
END
GO
```

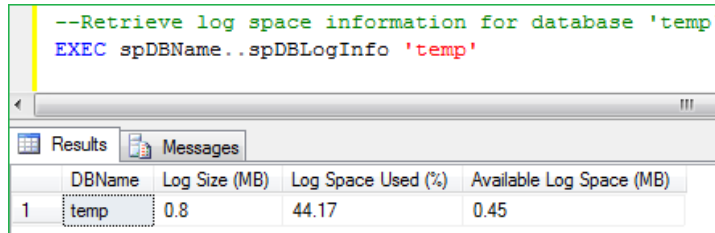
**Listing 2.3:** Creating a stored procedure for retrieving log space information for a given database.

Now, let's say we want to see log space information about the database 'temp'. The only thing we need to do is calling the stored procedure along with passing the database name as a parameter:

```
EXEC spDBName..spDBLogInfo 'temp';
GO
```

**Listing 2.4:** Executing the stored procedure created in 2.3.

For this example, this is what we get after executing the stored procedure:



	DBName	Log Size (MB)	Log Space Used (%)	Available Log Space (MB)
1	temp	0.8	44.17	0.45

**Figure 2.1:** Log Space Information for Sample Database Temp.

→ *Applies to: SQL Server 2000 or later.*

## ■ Handling Disk Space Issues During Heavy Index Rebuild Operations

(source: <http://aartemiou.blogspot.com/2012/11/handling-disk-space-issues-during-heavy.html>)

There are times where you need to massively rebuild indexes on some really large databases, after indicated by the [relevant analysis](#) (index fragmentation assessment).

**However, rebuilding indexes requires also the adequate amount of free disk space that will be used during the rebuild operation (mainly for the sorting process).**

Usually the required space is the size of the indexes to be rebuilt plus some more space (more information on Index Disk Space can be found [here](#)). For example, when you have a clustered index and the table size is 50 GB you will need somewhere between 50-55 GB of free disk space in order for the rebuild process to run properly.

Imagine a scenario where you need to rebuild 10 clustered indexes and each table's size is 50 GB. If you just schedule the job without having in mind disk space allocation you might find your machine running out of disk space very soon! This can have undesirable consequences to the O.S. and any other running processes (i.e. the index rebuild operation).

To this end, before deploying an index rebuild operation you first need to make sure that you have an adequate amount of free disk space.

But which disk drive should you monitor? *The TempDB drive? The drive where the user-database is stored onto?*

When you design an index rebuild operation, you typically have the option to sort the rebuild results in the "TempDB" system database. If you use this option then you will need to make sure that there is enough disk space on the disk onto which the TempDB database is located.

If you do not choose to sort the rebuild results in the "TempDB" database, then you will need to make sure that there is enough disk space on the disk onto which the user database is located. Of course, you should always have disk space available (always allow for some GB to be available) for TempDB as it is used in most of the SQL Server operations. However, in this case you need to focus on the drive where the database is located on.

Last but not least, a practice which I find useful because it helps avoiding disk space issues when running heavy index rebuild operations is the following:

After each large index rebuild you can include the following T-SQL statement:

```
DBCC SHRINKDATABASE (DBName, TRUNCATEONLY);  
GO
```

**Listing 2.5:** Shrinking a database's data/log file by releasing the free space at the end of the files.

From [Books Online](#), when the **TRUNCATEONLY** option is used, it "*releases all free space at the end of the file to the operating system but does not perform any page movement inside the file. The data file is shrunk only to the last allocated extent.*"

By the time there is no page movement inside the files, the space that will be used for the index rebuild, will be returned to the Operating System fast. The only prerequisite is that you run the shrink operation right after the index rebuild.

Let's conclude with pseudo code displaying how the rebuild process should look like:

```
REBUILD INDEX 1  
GO  
DBCC SHRINKDATABASE (DBName, TRUNCATEONLY);  
GO  
REBUILD INDEX 2  
GO  
DBCC SHRINKDATABASE (DBName, TRUNCATEONLY);  
GO  
REBUILD INDEX 3  
GO  
DBCC SHRINKDATABASE (DBName, TRUNCATEONLY);  
GO  
REBUILD INDEX N  
GO  
DBCC SHRINKDATABASE (DBName, TRUNCATEONLY);  
GO
```

**Listing 2.6:** Pseudocode for rebuilding indexes in large databases.



You can also include in the above logic additional maintenance tasks like update statistics, etc.

*Applies to: SQL Server 2000 or later.*

### ■ Executing Heavy Set-Based Operations Against VLDBs in SQL Server

(source: <http://aartemiou.blogspot.com/2011/05/executing-heavy-set-based-operations.html>)

At some time in the past, I had to design and execute some heavy set-based T-SQL operations against a large number of very large databases (VLDBs) within a SQL Server 2005 instance.

In this article, I am not discussing the debate of Row-Based vs. Set-Based processing, as I plan to write an article on this in the near future, but I am rather discussing some concerns that need to be taken into account when performing such operations. This article is related to the previous one.

First of all, in terms of hardware resources you will need enough RAM and of course, more than one processor (remember to check the Processor affinity setting in your SQL Server instance).

The most important aspect when executing such operations (set-based) against large databases is the log space. As the execution of the operation continues, a significant amount of transaction log space is being reserved thus requiring more storage for allocation. The reason this is happening is that the operation is not committed until it completes its execution against the entire data set. So, if your target database's recovery model is set to "FULL" it means that any changes performed by the set-based operation will be logged up to the latest detail thus requiring more storage.

So, how can you ensure that the set-based operation will be completed without any problems such as full disk spaces, etc. and thus avoid the possibility of having the operation abnormally terminated? Well, there are two approaches (if not more).

The first and more complex approach is to break-up the set-based operation thus targeting smaller data sets within the database. In my opinion however, this should be the very last option as "segmenting" the target data set is not a trivial task and can affect the integrity of the data.

The second approach, even simpler, **requires downtime**. The idea is:

**1. Set the database to single-user mode (this means downtime):**

```
USE master;  
GO  
ALTER DATABASE [Database_Name]  
SET SINGLE_USER  
WITH ROLLBACK IMMEDIATE;  
GO
```

**Listing 2.7:** Setting single-user access for a database.

### 2. Set the database's recovery model to SIMPLE (if previously was set to FULL):

```
ALTER DATABASE [Database_Name] SET RECOVERY SIMPLE;  
GO
```

**Listing 2.8:** Setting the recovery model of database to Simple.

### 3. Execute the set-based operation.

### 4. Set the database's recovery model back to FULL:

```
ALTER DATABASE [Database_Name] SET RECOVERY FULL;  
GO
```

**Listing 2.9:** Setting the recovery model of database to Full.

### 5. Set the database to multi-user mode (this means downtime):

```
USE master;  
GO  
ALTER DATABASE [Database_Name]  
SET MULTI_USER;  
GO
```

**Listing 2.10:** Setting multi-user access for a database.

When your database uses the Full Recovery Model, SQL Server preserves the transaction log until you back it up. By doing this, it allows you to recover the database to a specific point of time based on the entries in the transaction log. That's why when this recovery model is selected SQL Server logs everything in the database's log file. Of course this requires more disk space and slightly affects performance but it enables you to fully recover your database in the case of a disaster to the nearest point in time before the disaster.

When you use the Simple Recovery Model, SQL Server keeps only a minimal amount of information in the log file. Also the file is automatically truncated by SQL Server whenever the database reaches a transaction checkpoint. **Even though this is not the best choice for disaster recovery purposes, it allows the processing on the database to be faster.**

Someone might argue: "*why using the simple recovery model instead of the Bulk\_Logged?*" It is a fact that Simple and Bulk-Logged Recovery models indeed both allow high-performance bulk copy operations. However they have some differences. The Bulk-Logged Recovery Model minimally logs *bulk operations*. However it fully logs all other transactions. The Simple Recovery

Model does not backup the transaction log at all. It actually logs only minimal data related to the user that performs operations against the database. Also the Simple Recovery Model requires the least administration.

The reason I used the Simple Recovery Model in this case is that I just wanted to fully avoid the transaction log backup in order for executing the heavy set-based operations and switching back to the Full Recovery Model. Of course, as said above, the prerequisite is to have downtime and be the only user accessing the database (single\_user access). **If you use the above technique but you do not set the database to single\_user mode, you are risking not being able to recover your database to nearest point in time in the case of an unexpected failure. To this end, use the above technique with caution and always backup your data prior to using it!**

Generally, when it comes to selecting the recovery model for a proper backup strategy, especially on a Production Environment, you first need to analyze the business requirements that govern the target database(s) in order to make the right choice. For more information on SQL Server Recovery Models please visit the following [MDSN library article](#).

➔ ***Applies to: SQL Server 2000 or later.***

### ■ Summary

In this chapter, you learned how you can retrieve log space information for different databases hosted in SQL Server, as well as how to manage large log files in cases where you run heavy operations (i.e. the rebuild of large indexes). The next chapter talks about concurrency control in SQL Server (locking) and blocking.

## CHAPTER 3

### IN THIS CHAPTER:

- Monitoring Locking in SQL Server
- Updating SQL Server Tables Using Hints
- Table-Level Locking Hints in SQL Server

# Locking and Blocking

**The majority** of modern relational database management systems (RDBMSs) make use of lock-based concurrency. Lock-based concurrency is the approach based on which the Database Engine of a RDBMS ensures that no actions of committed transactions are lost. This is actually what **Locking** is.

SQL Server locks resources using different lock modes that determine how the resources can be accessed by concurrent transactions. SQL Server uses the following lock modes:

- Shared (S)
- Update (U)
- Exclusive (X)
- Intent
- Schema
- Bulk Update (BU)

In RDBMSs with lock-based concurrency, such as SQL Server, there are many cases where **blocking** can occur. Blocking takes place when one server process id (SPID) holds a lock on one resource and a second SPID tries to place a lock of a conflicting type on the same resource. Most of the times, each blocking incident does not take much time and it is a natural behavior of RDBMSs that use lock-based concurrency in order to help ensure data integrity.

There are sometimes however undesirable “blocking” incidents. Such incidents are:

- **Starvation:** It is the case where a transaction does not release a lock on a table/page thus forcing another transaction to wait indefinitely. SQL Server handles such issues with timeouts.
- **Deadlock:** It is the case where two transactions wait for each other in order to release their respective locks. SQL Server handles this by automatically selecting one of the two transactions and aborting the other along with rolling it back and sending an error message.

This chapter discusses how you can monitor locking and track blocking cases in SQL Server as well as other locking-related topics.

## ■ Monitoring Locking in SQL Server

(source: <http://aartemiou.blogspot.com/2013/09/monitoring-locking-in-sql-server.html>)

SQL Server 2005 (or later) has a specific dynamic management view (DMV) which provides detailed information regarding the active locks within the SQL Server instance, that is locks that have been already granted or they are waiting to be granted. The DMV is called: [sys.dm\\_tran\\_locks](#).

The following T-SQL query uses sys.dm\_tran\_locks in order to return useful information:

```
SELECT
resource_type,
resource_database_id,
(select [name] from master.sys.databases where database_id =resource_database_id)
as [DBName],
(case resource_type when 'OBJECT' then
object_name(resource_associated_entity_id,resource_database_id) else NULL end) as
ObjectName,
resource_associated_entity_id,
request_status, request_mode,request_session_id,
resource_description
FROM sys.dm_tran_locks;
GO
```

**Listing 3.1:** Retrieving locking information for a SQL Server instance.

A sample output of the above query would look like the one below:

	resource_type	resource_database_id	DBName	ObjectName	resource_associated_entity_id	request_status	request_mode	request_session_id	resource_description
1	DATABASE	4	msdb	NULL	0	GRANT	S	54	
2	DATABASE	8	AdventureWorks2012	NULL	0	GRANT	S	57	
3	DATABASE	8	AdventureWorks2012	NULL	0	GRANT	S	51	
4	DATABASE	8	AdventureWorks2012	NULL	0	GRANT	S	55	
5	OBJECT	1	master	sysobjvalues	60	GRANT	Sch-S	52	

**Figure 3.1:** Sample Query Output for Locking Information

An alternative way of monitoring the active locks is to use the “[Activity Monitor](#)” module in SQL Server Management Studio.

By joining the records returned by [sys.dm\\_tran\\_locks](#) and [sys.dm\\_os\\_waiting\\_tasks](#) DMVs you can get blocking information:

```

SELECT
t1.resource_type,
t1.resource_database_id,
(select [name] from master.sys.databases where database_id =resource_database_id)
as [DBName],
t1.resource_associated_entity_id,
(case resource_type when 'OBJECT' then
object_name(resource_associated_entity_id,resource_database_id) else NULL end) as
ObjectName,
t1.request_mode,
t1.request_session_id,
t1.blocking_session_id
FROM sys.dm_tran_locks as t1, sys.dm_os_waiting_tasks as t2
WHERE t1.lock_owner_address = t2.resource_address;
GO

```

**Listing 3.2:** Retrieving blocking information for a SQL Server instance.

The above two queries provide you with information on active locks and blocking cases and can help you identify scenarios that might need manual intervention (i.e. in cases of a bad database design and when the database is being accessed concurrently).

➔ *Applies to: SQL Server 2005 or later.*

## ■ Updating SQL Server Tables Using Hints

(source: <http://aartemiou.blogspot.com/2011/08/updating-sql-server-tables-without.html>)

Even though the SQL Server Database Engine automatically sets the best possible locking hints on the underlying database objects of the various T-SQL operations that executes, there are cases where we might need to manually control locking as a part of the business logic in our T-SQL script.

A popular locking hint is the [NOLOCK](#) as it can be used in environments with high concurrency. By using the NOLOCK hint, the transaction isolation level for the SELECT statement is READ UNCOMMITTED. Of course, this means that the query may see inconsistent data (dirty reads), that is data not yet committed, etc. The NOLOCK hint can only be used in SELECT statements.

*Even though the NOLOCK hint may seem handy in some cases, do not use it (or any other hint) unless you are sure that the issue cannot be resolved in any other way than using the hint and you deal with dirty reads.*

Now imagine the following scenario: You need to design a special UPDATE statement that will be updating an unknown number of records in a table which is being concurrently accessed by several other T-SQL statements (mostly UPDATE statements). *The above statement will be updating the table with non-critical information meaning that if it skips some records the first time, it can update them the second time and so on.*

If even one of the other UPDATE statements has locked a row that needs to be modified by your

UPDATE statement, this will cause the latter to wait (blocking). However, in the case where waiting is not a much "desired" option what should you do? In that case you could use the **READPAST** locking hint.

The **READPAST** locking hint when used, instructs the SQL Server Database Engine to skip row-level locks. This means that the UPDATE statement using READPAST will only update the table rows that are not locked by another operation. In the opposite case, the Database Engine would block the UPDATE statement's execution until the rest of the target rows' locks are released. A typical UPDATE statement with the READPAST locking hint would look like this:

```
UPDATE [TABLE_NAME] WITH (READPAST)
SET ...
WHERE ...
```

**Listing 3.3:** Sample code for updating a table using hints.

**\*Important:** Even if the above locking hint can become quite handy, as well as the rest of the locking hints, you always need to have in mind that you should use them very carefully as you might cause locking issues in the database. SQL Server Query Optimizer typically selects the best execution plan for a query, so it is not recommended for inexperienced developers and administrators to make use of the locking hints.

As a last note, the READPAST hint can only be specified in transactions operating at the READ COMMITTED or REPEATABLE READ isolation levels.

➔ *Applies to: SQL Server 2005 or later.*

### ■ Table-Level Locking Hints in SQL Server

(source: <http://aartemiou.blogspot.com/2009/01/table-level-locking-hints-in-sql-server.html>)

I was once asked by a friend, how we can control the level of locking in SQL Server when executing SELECT, UPDATE, INSERT and DELETE statements. In the beginning I was trying to avoid answering to him by strongly supporting the position that *SQL Server users should avoid using locking hints as the Database Engine can efficiently handle locking*. **When manually using hints, there is the danger of causing inconsistencies in your data**. However my friend insisted and so I had to answer and thus talk about locking hints.

Locking hints direct SQL Server to the type of locks to be used. Even though the SQL Server Query Optimizer automatically determines the best locking option when executing a statement, there are cases where after some serious testing a DBA or Database Developer might want to explicitly control the level of locking.

The available locking hints in SQL Server are the following:

### **HOLDLOCK**

Applies to: SELECT, UPDATE, INSERT, DELETE

### **NOLOCK**

Applies to: SELECT

### **PAGLOCK**

Applies to: SELECT, UPDATE, INSERT, DELETE

### **READCOMMITTED**

Applies to: SELECT, UPDATE, INSERT, DELETE

### **READPAST**

Applies to: SELECT, UPDATE, DELETE

### **READUNCOMMITTED**

Applies to: SELECT

### **REPEATABLEREAD**

Applies to: SELECT, UPDATE, INSERT, DELETE

### **ROWLOCK**

Applies to: SELECT, UPDATE, INSERT, DELETE

### **SERIALIZABLE**

Applies to: SELECT, UPDATE, INSERT, DELETE

### **TABLOCK**

Applies to: SELECT, UPDATE, INSERT, DELETE

### **TABLOCKX**

Applies to: SELECT, UPDATE, INSERT, DELETE

### **UPDLOCK**

Applies to: SELECT, UPDATE, INSERT, DELETE

### **XLOCK**

Applies to: SELECT, UPDATE, INSERT, DELETE

The syntax for using locking hints is very simple. You only have to add the expression **WITH (LOCKING\_HINT)** just right after the database table name (or table alias) referenced in your query.



An example of using the **NOLOCK** locking hint within a **SELECT** statement is the following:

```
SELECT column_name  
FROM table_name WITH (NOLOCK);  
GO
```

**Listing 3.4:** Using the NOLOCK table hint.

The following MSDN Library [article](#) fully describes the above locking hints.

→ *Applies to: SQL Server 2000 (partially), 2005 or later (fully).*

### ■ Summary

In this chapter, you learned what locking and blocking are, how you can update SQL Server tables with avoiding blocking, as well as how you can use table-level locking hints in SQL Server. The next chapter talks about one of the system databases that are shipped with SQL Server: **tempdb**.

## CHAPTER 4

# TempDB

### **IN THIS CHAPTER:**

- TempDB Growth
- Where are Temporary Tables Stored in SQL Server?

**The tempdb** system database in SQL Server is one of the databases that are shipped with every SQL Server installation and it is used for specific tasks as a temporary storage. Examples of when tempdb is used are:

- Use of local/global temporary tables, stored procedures and variables.
- Use of cursors.
- When the SORT\_IN\_TEMPDB option is used during the creation or rebuild of indexes.
- Use of local/global stored procedures.
- When row versions are generated by different transactions.

TempDB is protected with the use of certain restrictions. These are:

- You cannot drop it.
- You cannot back it up/restore it.
- You cannot change the database owner (it is always the dbo).
- You cannot create a database snapshot.
- You cannot drop the guest user.
- You cannot add filegroups.
- You cannot change the collation (default collation is server's collation).
- You cannot take the database offline.
- You cannot set the database or primary filegroup to READ\_ONLY.
- You cannot run DBCC CHECKALLOC and DBCC CHECKCATALOG.
- You cannot rename the database or its primary filegroup.
- You cannot remove the primary filegroup, data or log file.
- TempDB cannot participate in database mirroring.
- You cannot enable change data capture on TempDB.

This chapter includes articles from my blog having to do with maintaining the good health and performance of the tempdb database as well as retrieving information from tempdb data structures.

## ■ TempDB Growth

(source: <http://aartemiou.blogspot.com/2009/05/tempdb-growth.html>)

As its names implies, the `tempdb` database contains temporary data that are created and maintained during SQL Server operations. Such data may include: temporary user objects (i.e. temporary tables, cursors), row versions (i.e. those that come up from online index operations), and other internal objects that are created by SQL Server Database Engine.

The tempdb is global on a SQL Server Instance, that is available to all the users/databases on a SQL Server Instance, so in the cases where an instance might contain a large number of databases resulting to a large number of operations that use temporary data, this might have as an effect the tempdb size to increase rapidly.

To view the tempdb size and growth parameters, you can use SSMS GUI or run the following query:

```
SELECT
    [name] as [FileName],
    (cast ((size*1.0/128) as decimal (18,2))) AS [FileSizeinMB],
    (case growth when 0 then 'Autogrowth: Off' else 'Autogrowth: On' end)
as [AutogrowthMode],
    (case max_size when -1 then 'Unrestricted' else cast(cast((max_size/1024.0*8)
as int) as varchar(15)) end) as [MaxSizeMB],
    (case is_percent_growth when 1 then (CAST(growth as varchar(20)) + ' %') else
    (CAST(cast((growth/1024.0*8) as int) as varchar(20)) + ' MB') end) as
GrowthValue
FROM tempdb.sys.database_files;
GO
```

**Listing 4.1:** Retrieving tempdb data/log file size information

So, how can you shrink the tempdb database and limit its size?

Well, there is more than one option for doing this. The following [KB article](#) describes three different methods for shrinking tempdb. In summary, these are:

### Method 1

Altering the tempdb file size with the "ALTER DATABASE [tableName] MODIFY FILE" command

### Method 2

Using the "DBCC SHRINKDATABASE" command

### Method 3

Using the "DBCC SHRINKFILE" command

➔ *Applies to: SQL Server 2005 or later.*

## ■ Where are Temporary Tables Stored in SQL Server?

(source: <http://aartemiou.blogspot.com/2009/04/where-are-temporary-tables-stored-in.html>)

As its name implies, the `tempdb` database contains temporary data that is created and maintained by the Database Engine during SQL Server operations.

There are many cases where we need to create temporary tables in SQL Server and for various reasons such as:

- Breaking the logic of a large and complex SQL Statement in smaller portions of code.
- Increase the performance of a SQL query, etc.

But what types of temporary tables does SQL Server provide and what is the meaning of each type? Last but not least, where are temporary tables stored and how can we get schema information about them?

There are two types of temporary tables:

- Local temporary tables:
  - Only available to the current connection to the database for the current login.
  - They are dropped when the connection is closed.
- Global temporary tables:
  - Available to any connection upon their creation.
  - They are dropped when the last connection using them is closed.

### Code Example – Syntax for Creating a Local Temporary Table:

```
CREATE TABLE #table_name
(
    column_name1 [DATATYPE]
    column_name2 [DATATYPE]
    ...
    column_nameN [DATATYPE]
);
GO
```

**Listing 4.2:** Creating a local temporary table.

### Code Example – Syntax for Creating a Global Temporary Table:

```
CREATE TABLE ##table_name
(
    column_name1 [DATATYPE]
    column_name2 [DATATYPE]
    ...
    column_name [DATATYPE]
);
GO
```

**Listing 4.3:** Creating a global temporary table.

So, consider as an example that you create the following temporary table:

```
CREATE TABLE #temp_table
(
    id INT,
    name VARCHAR(50)
);
GO
```

**Listing 4.4:** Creating a sample temporary table.

Then let's say you want to find schema information regarding the above table. Where can you find this information in SQL Server?

The answer is that temporary tables (local and global) are stored in the **tempDB** database.

So, if you want to find schema information about the temporary table named **temp\_table** you can use the following queries:

```
--Query 1(a): Get the exact name of the temporary table you are looking for
DECLARE @table_name AS VARCHAR(300);

SET @table_name =
(
    SELECT TOP 1 [name]
    FROM tempdb..sysobjects
    WHERE name LIKE '#temp_table%'
);
```

**Listing 4.5:** Getting the exact name of the temporary table as stored in tempdb.

**Explanation:** When you declare a temporary table, SQL Server adds some additional characters on its name in order to provide a unique system name for it and then it stores it in **tempDB** in the **sysobjects** table. Even though you can query the temporary table with its logical name, internally it is known with the exact name that the SQL Server Database Engine has set. To this end, you need to execute the above query for finding the exact name of the temporary table.

```
--Query 1(b): Get column information for the temporary table
-- by using the sp_columns stored procedure
EXEC tempdb..sp_columns @table_name;
GO
```

**Listing 4.6:** Getting column information for the temp table in tempdb.

**Explanation:** The **sp\_columns** stored procedure returns column information for the specified tables or views that can be queried in the current environment.

➔ *Applies to: SQL Server 2000 or later.*

## ■ Summary

In this chapter you learned about the usage and restrictions of the tempdb database. You learned how to monitor and control its growth and how to retrieve information about the objects that are getting stored in the database.



# List of Listings

<b>Listing 1.1:</b> Retrieving index information using sp_helpindex	12
<b>Listing 1.2:</b> Retrieving index fragmentation in SQL Server 2000	13
<b>Listing 1.3:</b> Retrieving index fragmentation in SQL Server 2005 or later	14
<b>Listing 1.4:</b> Pseudocode for assessing index fragmentation	15
<b>Listing 1.5:</b> Rebuilding indexes in SQL Server 2005 or later	15
<b>Listing 1.6:</b> Rebuilding indexes in SQL Server 2000	15
<b>Listing 1.7:</b> Reorganizing indexes in SQL Server 2005 or later	16
<b>Listing 1.8:</b> Reorganizing indexes in SQL Server 2000	16
<b>Listing 1.9:</b> Rebuilding all indexes for a database in SQL Server 2000	17
<b>Listing 1.10:</b> Rebuilding all indexes for a database in SQL Server 2005 or later	18
<b>Listing 2.1:</b> Retrieving log space information	21
<b>Listing 2.2:</b> Storing log space information in a temporary table	21
<b>Listing 2.3:</b> Creating a stored procedure for retrieving log space information for a given database	22
<b>Listing 2.4:</b> Executing the stored procedure created in 2.3	23
<b>Listing 2.5:</b> Shrinking a database's data/log file by releasing the free space at the end of the files	24
<b>Listing 2.6:</b> Pseudocode for rebuilding indexes in large databases	24
<b>Listing 2.7:</b> Setting single-user access for a database	26
<b>Listing 2.8:</b> Setting the recovery model of database to Simple	26
<b>Listing 2.9:</b> Setting the recovery model of database to Full	26
<b>Listing 2.10:</b> Setting multi-user access for a database	26
<b>Listing 3.1:</b> Retrieving locking information for a SQL Server instance	29
<b>Listing 3.2:</b> Retrieving blocking information for a SQL Server instance	30
<b>Listing 3.3:</b> Sample code for updating a table using hints	31
<b>Listing 3.4:</b> Using the NOLOCK table hint	33
<b>Listing 4.1:</b> Retrieving tempdb data/log file size information	35
<b>Listing 4.2:</b> Creating a local temporary table	36
<b>Listing 4.3:</b> Creating a global temporary table	36
<b>Listing 4.4:</b> Creating a sample temporary table	37
<b>Listing 4.5:</b> Getting the exact name of the temporary table as stored in tempdb	37
<b>Listing 4.6:</b> Getting column information for the temp table in tempdb	37

# List of Figures

<b>Figure 2.1:</b> Log Space Information for Sample Database Temp	23
<b>Figure 3.1:</b> Sample Query Output for Locking Information	29



