THE EXPERT'S VOICE® IN SQL SERVER

Expert SQL Server In-Memory OLTP

REVOLUTIONIZING OLTP PERFORMANCE IN SQL SERVER

Dmitri Korotkevitch

Apress®

Expert SQL Server In-Memory OLTP

Dmitri Korotkevitch

Apress®

Expert SQL Server In-Memory OLTP

ISBN-13 (pbk): 978-1-4842-1137-3

ISBN-13 (electronic): 978-1-4842-1136-6

Managing Director: Welmoed Spahr
Lead Editor: Jonathan Gennick
Development Editor: Douglas Pundick
Technical Reviewer: Sergey Olontsev
Editorial Board: Steve Anglin, Mark Beckner, Gary Cornell, Louise Corrigan, Jim DeWolf, Jonathan Gennick, Robert Hutchinson, Michelle Lowman, James Markham, Susan McDermott, Matthew Moodie, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Gwenan Spearing, Matt Wade, Steve Weiss
Coordinating Editor: Jill Balzano
Copy Editor: Mary Behr
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

To all my friends from the SQL Server community and outside of it

Contents at a Glance

About the Author xiii

About the Technical Reviewer xv

Acknowledgments xvii

Introduction xix

■ Chapter 1: Why In-Memory OLTP? 1

■ Chapter 2: In-Memory OLTP Objects 7

■ Chapter 3: Memory-Optimized Tables 27

■ Chapter 4: Hash Indexes 39

■ Chapter 5: Nonclustered Indexes 61

■ Chapter 6: In-Memory OLTP Programmability 79

■ Chapter 7: Transaction Processing in In-Memory OLTP 103

■ Chapter 8: Data Storage, Logging, and Recovery 121

■ Chapter 9: Garbage Collection 135

■ Chapter 10: Deployment and Management 147

■ Chapter 11: Utilizing In-Memory OLTP 169

■ Appendix A: Memory Pointers Management 209

■ Appendix B: Page Splitting and Page Merging in Nonclustered Indexes 213

Appendix C: Analyzing the States of Checkpoint File Pairs 219

Appendix D: In-Memory OLTP Migration Tools 233

Index 245

Contents

About the Author xiii

About the Technical Reviewer xv

Acknowledgments xvii

Introduction xix

Chapter 1: Why In-Memory OLTP? 1

Background 1

In-Memory OLTP Engine Architecture 3

Summary 5

Chapter 2: In-Memory OLTP Objects 7

Preparing a Database to Use In-Memory OLTP 7

Creating Memory-Optimized Tables 8

Working with Memory-Optimized Tables 11

In-Memory OLTP in Action: Resolving Latch Contention 16

Summary 24

Chapter 3: Memory-Optimized Tables 27

On-Disk vs. Memory-Optimized Tables 27

Introduction to the Multiversion Concurrency Control 31

Data Row Format 33

Native Compilation of Memory-Optimized Tables 35

Memory-Optimized Tables: Surface Area and Limitations 36
Supported Data Types 36
Constraints and Table Features 37
Database-Level Limitations 37
High Availability Technologies Support 38
Summary 38

Chapter 4: Hash Indexes 39

Hashing Overview 39
Much Ado About Bucket Count 40
Bucket Count and Performance 41
Choosing the Right Bucket Count 46
Changing the Bucket Count in the Index 47
Hash Indexes and SARGability 48
Statistics on Memory-Optimized Tables 52
Summary 58

Chapter 5: Nonclustered Indexes 61

Working with Nonclustered Indexes 61
Creating Nonclustered Indexes 62
Using Nonclustered Indexes 62
Nonclustered Indexes Internals 67
Bw-Tree Overview 67
Index Pages and Delta Records 69
Obtaining Information About Nonclustered Indexes 71
Hash Indexes vs. Nonclustered Indexes 74
Summary 78

Chapter 6: In-Memory OLTP Programmability 79

Native Compilation 79

Natively Compiled Stored Procedures 84

Creating Natively Compiled Stored Procedures 84

Supported T-SQL Features 86

Atomic Blocks 88

Optimization of Natively Compiled Stored Procedures 91

Interpreted T-SQL and Memory-Optimized Tables 92

Performance Comparison 92

Memory-Optimized Table Types and Variables 99

Summary 101

Chapter 7: Transaction Processing in In-Memory OLTP 103

ACID, Transaction Isolation Levels, and Concurrency Phenomena Overview 103

Transaction Isolation Levels in In-Memory OLTP 106

Cross-Container Transactions 112

Transaction Lifetime 114

Summary 119

Chapter 8: Data Storage, Logging, and Recovery 121

Data Storage 121

Checkpoint File Pairs States 123

Transaction Logging 128

Recovery 131

Summary 133

Chapter 9: Garbage Collection 135

Garbage Collection Process Overview 135

Garbage Collection-Related Data Management Views 140

Exploring the Garbage Collection Process 140

Summary 145

Chapter 10: Deployment and Management 147

Hardware Considerations 147

CPU 148

I/O Subsystem 148

Memory 149

Administration and Monitoring Tasks 151

Limiting the Amount of Memory Available to In-Memory OLTP 151

Monitoring Memory Usage for Memory-Optimized Tables 153

Monitoring In-Memory OLTP Transactions 157

Collecting Execution Statistics for Natively Compiled Stored Procedures 159

Metadata Changes and Enhancements 162

Catalog Views 162

Data Management Views 162

Extended Events and Performance Counters 165

Summary 168

Chapter 11: Utilizing In-Memory OLTP 169

Design Considerations for the Systems Utilizing In-Memory OLTP 169

Addressing In-Memory OLTP Limitations 171

8,060-Byte Maximum Row Size Limit 171

Lack of Uniqueness and Foreign Key Constraints 176

Case-Sensitivity Binary Collation for Indexed Columns 182

Thinking Outside the In-Memory Box 185
Importing Batches of Rows from Client Applications 185
Using Memory-Optimized Objects as Replacements for Temporary and Staging Tables 188
Using In-Memory OLTP as Session - or Object State-Store 196
Using In-Memory OLTP in Systems with Mixed Workloads 203
Summary 207
Appendix A: Memory Pointers Management 209
Memory Pointers Management 209
Summary 211
Appendix B: Page Splitting and Page Merging in Nonclustered Indexes 213
Nonclustered Indexes Internal Maintenance 213
Page Splitting 213
Page Merging 215
Summary 217
Appendix C: Analyzing the States of Checkpoint File Pairs 219
Sys.db_dm_xtp_checkpoint_files View 219
The Lifetime of Checkpoint File Pairs 220
Summary 232
Appendix D: In-Memory OLTP Migration Tools 233
Management Data Warehouse Enhancements 233
Memory Optimization and Native Compilation Advisors 242
Summary 244
Index 245

About the Author

Dmitri Korotkevitch is a Microsoft SQL Server MVP and a Microsoft Certified Master (SQL Server 2008) with 20 years of IT experience including over 15 years of experience working with Microsoft SQL Server as an application and database developer, database administrator, and database architect. Dmitri specializes in the design, development, and performance tuning of complex OLTP systems that handle thousands of transactions per second around the clock.

Dmitri regularly speaks at various Microsoft and SQL PASS events, and he provides SQL Server training to clients around the world. He blogs at http://aboutsqlserver.com, and he can be reached at dk@aboutsqlserver.com.

About the Technical Reviewer

Sergey Olontsev have been working with SQL Server for more than 10 years both as a database administrator and a developer, focusing on high availability and disaster recovery solutions, ETL, troubleshooting and performance tuning, and developing highly scalable solutions. He has MCM certification and has won an MVP award for SQL Server; he is also a regular speaker at SQL Server user group meetings, SQL Saturday events, and other conferences across Russia, Europe, and the USA. Currently Sergey is living in Moscow, Russia and is working for Kaspersky Lab, where he is responsible for developing and supporting several large internal databases. Maximum performance and minimum latency while processing data are major goals in his current job, so his company started using In-Memory OLTP to implement it in the most critical parts.

Acknowledgments

First and foremost, I would like to thank my family for their patience, understanding, and continuous support. It would be impossible for me to write this book without it!

I am very grateful to my technical reviewer, Sergey Olontsev. His comments and suggestions were enormously helpful and dramatically improved the quality of the book.

The same goes to the entire Apress team and especially to Jill Balzano, Mary Behr, and Douglas Pundick. Special thanks to Jonathan Gennick, who convinced me to write this book in the first place. We had very productive lunch, Jonathan, had we not? ☺

Finally, I would like to thank my colleagues Vladimir Zatuliveter for his help with the Chapter 11 code and Jason Vorbeck for providing me the testing environment I used during my work on the book.

Introduction

Writing is an interesting process. You feel happy and energized when looking at the blank Microsoft Word document with the *Chapter 1* title. The energy and happiness, however, quickly disappear and are replaced by tiredness and the constant pressure of deadlines. Finally, when the book is done, you feel extremely exhausted and promise yourself that you will *never ever* do it again.

It does not take long, however, to begin missing the process, pressure, and sleepless nights. So when Jonathan Gennick from Apress asked me to consider writing a book on In-Memory OLTP, it was an easy sell for him. After all, In-Memory OLTP is a fascinating subject, and it changes the way you design OLTP systems. I was really disappointed that I was unable to dive deeper into it during my work on my *Pro SQL Server Internals* book.

The Microsoft implementation of in-memory data is hardly the first solution on the market. Nevertheless, it has several key differences from competitors' implementations. The biggest are the level of integration it provides with the classic Database Engine and its simplicity for the end users. You can move data into memory and start using it with just a handful of mouse clicks.

I would consider this simplicity, however, a double-edged sword. While it can significantly reduce technology adoption time and cost, it can also open the door to incorrect decisions and suboptimal implementations. As with any other technology, In-Memory OLTP has been designed for a specific set of tasks, and it can hurt performance of the systems when implemented incorrectly. Neither is it a "*set it and forget it*" type of solution; you have to carefully plan it before and maintain it after the deployment.

In-Memory OLTP is a great technology and it can dramatically improve the performance of systems. Nevertheless, you need to understand how it works under the hood to get the most from it. The goal I set for this book is to provide you with such an understanding. I will explain the internals of the In-Memory OLTP Engine and its components. I believe that knowledge is the cornerstone in successful In-Memory OLTP implementations and it will help you to make educated decisions on how and when to use the technology.

If you read my *Pro SQL Server Internals* book, you will notice some familiar content from there. However, this book is a much deeper dive into In-Memory OLTP and you will find plenty of new topics covered. You will also learn how to address some of In-Memory OLTP's limitations and how to benefit from it in existing systems when those limitations make in-memory migration cost ineffective.

I want to reiterate that this book is covering In-Memory OLTP in SQL Server 2014. Even though the core implementation principles will remain the same in SQL Server 2016 and future SQL Server releases, you should expect significant improvements in them. In-Memory OLTP is one of the flagship SQL Server technologies, and Microsoft is fully committed to it and is investing a large amount of engineering resources for this product.

Finally, I would like to thank you again for choosing this book and for your trust in me. I hope that you will enjoy reading it as much as I enjoyed writing it.

How This Book Is Structured

This book consists of 11 chapters and structured in the following way:

- **Chapter 1** and **Chapter 2** are the introductory chapters, which will provide you the overview of technology and show how In-Memory OLTP objects work together.
- **Chapter 3**, **Chapter 4**, and **Chapter 5** explain how In-Memory OLTP stores and works with data and indexes in memory.
- **Chapter 6** talks about native compilation and the programmability aspect of the technology.
- **Chapter 7** explains how In-Memory OLTP handles concurrency in a multi-user environment.
- **Chapter 8** demonstrates how In-Memory OLTP persists data on disk and how it works with the transaction log.
- **Chapter 9** covers the In-Memory OLTP garbage collection process.
- **Chapter 10** discusses the best practices for In-Memory OLTP deployments and shows how to perform common database administration tasks related to In-Memory OLTP.
- **Chapter 11** demonstrates how to address some of the In-Memory OLTP surface area limitations and how to benefit from In-Memory OLTP components without moving data into memory.

The book also includes four appendixes:

- **Appendix A** explains how In-Memory OLTP works with memory pointers in a multi-user environment.
- **Appendix B** covers how page splitting and merging processes are implemented.
- **Appendix C** shows you how to analyze the state of checkpoint file pairs and navigates you through their lifetime.
- **Appendix D** discusses SQL Server tools and wizards that can simplify In-Memory OLTP migration.

Downloading the Code

You can download the code used in this book from the Source Code section of the Apress web site (`www.apress.com`) or from the Publications section of my blog (`http://aboutsqlserver.com`). The source code consists of a SQL Server Management Studio solution, which includes a set of projects (one per chapter). Moreover, it includes several .Net C# projects, which provide the client application code used in the examples in Chapters 2 and 11.

I have tested all of the scripts in an environment with 3GB of RAM available to SQL Server. In some cases, if you have less memory available, you will need to reduce amount of test data generated by some of the scripts. You can also consider dropping some of the unused test tables to free up more memory.

Contacting the Author

You can visit my blog at http://aboutsqlserver.com or email me at dk@aboutsqlserver.com. As usual, I will be happy to answer any questions you have.

CHAPTER 1

Why In-Memory OLTP?

This introductory chapter explains the importance of in-memory databases and the problems they address. It provides an overview of the Microsoft In-Memory OLTP implementation (code name *Hekaton*) and its design goals. Finally, this chapter discusses the high-level architecture of the In-Memory OLTP Engine and how it is integrated into SQL Server.

Background

Way back when SQL Server and other major databases were originally designed, hardware was very expensive. Servers at that time had just one or very few CPUs, and a small amount of installed memory. Database servers had to work with data that resided on disk, loading it into memory on demand.

The situation has changed dramatically since then. During the last 30 years, memory prices have dropped by a factor of 10 every 5 years. Hardware has become more affordable. It is now entirely possible to buy a server with 32 cores and 1TB of RAM for less than $50,000. While it is also true that databases have become larger, it is often possible for *active* operational data to fit into the memory.

Obviously, it is beneficial to have data cached in the buffer pool. It reduces the load on the I/O subsystem and improves system performance. However, when systems work under a heavy concurrent load, it is often not enough. SQL Server manages and protects page structures in memory, which introduces large overhead and does not scale well. Even with row-level locking, multiple sessions cannot modify data on the same data page simultaneously and must wait for each other.

Perhaps the last sentence needs to be clarified. Obviously, multiple sessions can modify data rows on the same data page, holding exclusive (X) locks on different rows simultaneously. However, they cannot update *physical* data page and row objects simultaneously because it could corrupt the in-memory page structure. SQL Server addresses this problem by protecting pages with *latches*. Latches work in a similar manner to locks, protecting internal SQL Server data structures on the physical level by serializing access to them, so only one thread can update data on the data page in memory at any given point of time.

In the end, this limits the improvements that can be achieved with the current database systems architecture. Although you can scale hardware by adding more CPUs and cores, that serialization quickly becomes a bottleneck and a limiting factor in

improving system scalability. Likewise, you cannot improve performance by increasing the CPU clock speed because the silicon chips would melt down. Therefore, the only feasible way to improve database system performance is by reducing the number of CPU instructions that need to be executed to perform an action.

Unfortunately, code optimization is not enough by itself. Consider the situation where you need to update a row in a table. Even when you know the clustered key value, that operation needs to traverse the clustered index tree, obtaining latches and locks on the data pages and a row. In some cases, it needs to update nonclustered indexes, obtaining the latches and locks there. All of that generates log records and requires writing them and the dirty data pages to disk.

All of those actions can lead to a hundred thousand or even millions of CPU instructions to execute. Code optimization can help reduce this number to some degree, but it is impossible to reduce it dramatically without changing the system architecture and the way the system stores and works with data.

These trends and architectural limitations led the Microsoft team to the conclusion that a true in-memory solution should be built using different design principles and architecture than the classic SQL Server Database Engine. The original concept was proposed at the end of 2008, serious planning and design started in 2010, and actual development began in 2011.

The main goal of the project was to build a solution that will be 100 times faster than the existing SQL Server Engine, which explains the code name *Hekaton* (Greek for *one hundred*). This goal has yet to be achieved; however, the first production release of In-Memory OLTP can provide 20X-40X performance improvements in certain scenarios.

It is also worth mentioning that the Hekaton design has been targeted towards the OLTP workload. As all of us know, specialized solutions designed for particular tasks and workloads usually outperform general-purpose systems in the targeted areas. The same is true for In-Memory OLTP. It shines with large and very busy OLTP systems that support hundreds or even thousands of concurrent users. At the same time, In-Memory OLTP is not necessarily the best choice for Data Warehouse workload, where other SQL Server components could outperform it.

In-Memory OLTP has been designed with the following goals:

- **Optimize data storage for main memory:** Data in In-Memory OLTP is not stored on on-disk data pages nor does it mimic an on-disk storage structure when loaded into memory. This permits the elimination of the complex buffer pool structure and the code that manages it. Moreover, indexes are not persisted on disk, and they are re-created upon startup when the data from memory-resident tables is loaded into memory.

- **Eliminate latches and locks:** All In-Memory OLTP internal data structures are latch- and lock-free. In-Memory OLTP uses a new multiversion concurrency control to provide transaction consistency. From a user standpoint, it behaves similar to the regular `SNAPSHOT` transaction isolation level; however, it does not use a locking or `tempdb` version store under the hood. This schema allows multiple sessions to work with the same data without locking and blocking each other and improves the scalability of the system, allowing it to fully utilize modern multi-CPU/multi-core hardware.

- **Use native compilation:** T-SQL is an interpreted language that provides great flexibility at the cost of CPU overhead. Even a simple statement requires hundreds of thousands of CPU instructions to execute. The In-Memory OLTP Engine addresses this by compiling row-access logic and stored procedures into native machine code.

The In-Memory OLTP Engine is fully integrated in the SQL Server Engine, which is the key differentiator of the Microsoft implementation compared to other in-memory database solutions. You do not need to perform complex system refactoring, splitting data between in-memory and conventional database servers, or moving all of the data from the database into memory. You can separate in-memory and disk data on a table-by-table basis, which allows you to move active operational data into memory, keeping other tables and historical data on disk. In some cases, that migration can even be done transparently to client applications.

It sounds too good to be true and, unfortunately, there are still plenty of roadblocks that you may encounter when working with this technology. The first release of In-Memory OLTP supports just a subset of the SQL Server data types and features, which often requires you to perform code and schema refactoring to utilize it. We will discuss those limitations later in the book; however, you need to know that Microsoft is fully committed to this project. You can expect that future versions of In-Memory OLTP will have a bigger surface area and fewer restrictions compared to the initial release. In fact, you can already see the changes in the CTP releases of SQL Server 2016 and in SQL Databases in Microsoft Azure.

In-Memory OLTP Engine Architecture

In-Memory OLTP is fully integrated into SQL Server, and other SQL Server features and client applications can access it transparently. Internally, however, it works and behaves very differently than the SQL Server Storage Engine. Figure 1-1 shows the architecture of the SQL Server Engine including the In-Memory OLTP components.

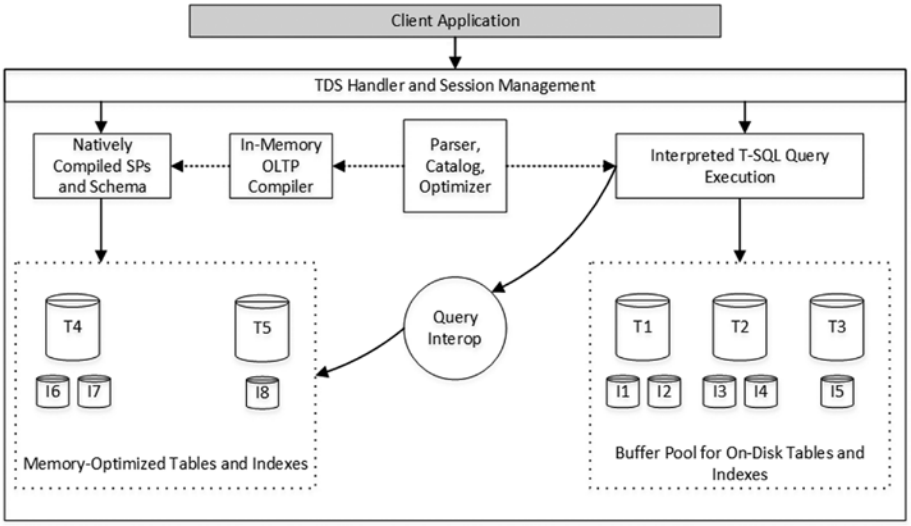


Figure 1-1. *SQL Server Engine architecture*

In-Memory OLTP stores the data in **memory-optimized tables**. These tables reside completely in memory and have a different structure compared to the classic **on-disk tables**. With one small exception, memory-optimized tables do not store data on the data pages linking the rows together through the chains of memory pointers. It is also worth noting that memory-optimized tables do not share memory with on-disk tables and live outside of the buffer pool.

■ **Note** We will discuss memory-optimized tables in detail in Chapter 3.

There are two ways the Database Engine can work with memory-optimized tables. The first is the **interop engine**. It allows you to reference memory-optimized tables from interpreted T-SQL code. The data location is transparent to the queries; you can access memory-optimized tables, join them together and with on-disk tables, and work with them in the usual way. Most T-SQL features and language constructs are supported in this mode.

You can also access and work with memory-optimized tables using **natively compiled stored procedures**. You can define those procedures similarly to the regular T-SQL stored procedures using several additional language constructs introduced by In-Memory OLTP.

Natively compiled stored procedures have been compiled into machine code and loaded into SQL Server process memory. Those procedures can introduce significant performance improvements compared to the interop engine; however, they support just a limited set of T-SQL constructs and can access only memory-optimized tables.

■ **Note** We will discuss natively compiled stored procedures in Chapter 6.

Now, it's time to see how In-Memory OLTP components work together, which we will do in the next chapter.

Summary

In-Memory OLTP has been designed using different design principles and architecture than the classic SQL Server Engine. It is a specialized product targeted towards the OLTP workload and can provide up to 20X-40X performance improvements in certain scenarios. Nevertheless, it is fully integrated into the SQL Server Database Engine. The data storage is transparent to the client applications, which do not require any code changes if they use the features supported by In-Memory OLTP.

The data from memory-optimized tables is stored in memory separately from the buffer pool. All of the In-Memory OLTP data structures are completely latch- and lock-free, which allows scaling the systems by adding more CPUs to the servers.

In-Memory OLTP uses native compilation to the machine code for any row-access logic. Moreover, it allows performing native compilation of the stored procedures, which dramatically increase their performance.

The first release of In-Memory OLTP has a large number of limitations; however, Microsoft is fully committed to the project and will address these limitations in future In-Memory OLTP releases.

CHAPTER 2

In-Memory OLTP Objects

This chapter provides a high-level overview of In-Memory OLTP objects. It shows how to create databases with an In-Memory OLTP filegroup, and how to define memory-optimized tables and access them through the interop engine and natively compiled stored procedures. Finally, this chapter demonstrates performance improvements that can be achieved with the In-Memory OLTP Engine when a large number of concurrent sessions insert the data into the database and latch contention becomes the bottleneck.

Preparing a Database to Use In-Memory OLTP

The In-Memory OLTP Engine has been fully integrated into the Enterprise Edition of SQL Server and is always installed with the product. It requires the 64-bit version of SQL Server; it is not supported in the 32-bit version. You do not need to install any additional packages nor perform any configuration changes on the SQL Server level in order to use it as long as you are using the correct version and edition of the product. However, any database that utilizes In-Memory OLTP objects should have a separate filegroup to store memory-optimized data.

You can create this filegroup at database creation time or alter an existing database and add the filegroup using the `CONTAINS MEMORY_OPTIMIZED_DATA` keyword.

Listing 2-1 shows the first example. The `FILENAME` property of the filegroup specifies the folder in which In-Memory OLTP files would be located.

Listing 2-1. Creating a Database with the In-Memory OLTP Filegroup

```
create database InMemoryOLTPDemo
on primary
(
    name = N'InMemoryOLTPDemo'
    ,filename = N'M:\Data\InMemoryOLTPDemo.mdf'
),
filegroup HKData CONTAINS MEMORY_OPTIMIZED_DATA
(
    name = N'InMemory_OLTP_Data'
    ,filename = N'H:\HKData\InMemory_OLTP_Data'
),
```

```
filegroup LOGDATA
(name = N'LogData1', filename = N'M:\Data\LogData1.ndf'),
(name = N'LogData2', filename = N'M:\Data\LogData2.ndf'),
(name = N'LogData3', filename = N'M:\Data\LogData3.ndf'),
(name = N'LogData4', filename = N'M:\Data\LogData4.ndf'),
log on
(
    name = N'InMemoryOLTPDemo_log'
    ,filename = N'L:\Log\InMemoryOLTPDemo_log.ldf'
)
```

Under the hood, In-Memory OLTP utilizes a streaming mechanism based on `FILESTREAM` technology. While coverage of `FILESTREAM` is outside the scope of this book, I would like to mention that it is optimized for sequential I/O access. In fact, In-Memory OLTP does not use random I/O access at all. It uses sequential append-only writes during a normal workload and sequential reads on the database startup and recovery stages. You should keep this behavior in mind and place In-Memory OLTP filegroups into the disk arrays optimized for sequential performance.

Lastly, similar to `FILESTREAM` filegroups, the In-Memory OLTP filegroup can include multiple containers placed on the different disk arrays.

■ **Note** You can read more about `FILESTREAM` at `http://technet.microsoft.com/en-us/library/gg471497.aspx`.

I will discuss the best practices in hardware and SQL Server configurations in Chapter 10.

Unfortunately, SQL Server 2014 does not allow you to remove an In-Memory OLTP filegroup from the database even after you drop all memory-optimized tables and objects. This limitation prevents you from restoring the database on non-Enterprise editions of SQL Server if such need ever arises.

■ **Tip** You can move a database between different editions of SQL Server as long as you do not use any features incompatible with the new (lower) edition. You can read more about it at `https://msdn.microsoft.com/en-us/library/cc280724.aspx`.

Creating Memory-Optimized Tables

Syntax-wise, creation of memory-optimized tables is very similar to on-disk tables. You can use the regular `CREATE TABLE` statement specifying that the table is memory-optimized.

The code in Listing 2-2 creates three memory-optimized tables in the database. Please ignore all unfamiliar constructs; I will discuss them in detail later.

Listing 2-2. Creating Memory-Optimized Tables

```
create table dbo.WebRequests_Memory
(
    RequestId int not null identity(1,1)
        primary key nonclustered
        hash with (bucket_count=1000000),
    RequestTime datetime2(4) not null
        constraint DEF_WebRequests_Memory_RequestTime
        default sysutcdatetime(),
    URL varchar(255) not null,
    RequestType tinyint not null, -- GET/POST/PUT
    ClientIP varchar(15)
        collate Latin1_General_100_BIN2 not null,
    BytesReceived int not null,

    index IDX_RequestTime nonclustered(RequestTime)
)
with (memory_optimized=on, durability=schema_and_data);

create table dbo.WebRequestHeaders_Memory
(
    RequestHeaderId int not null identity(1,1)
        primary key nonclustered
        hash with (bucket_count=5000000),
    RequestId int not null,
    HeaderName varchar(64) not null,
    HeaderValue varchar(256) not null,

    index IDX_RequestID nonclustered hash(RequestID)
    with (bucket_count=1000000)
)
with (memory_optimized=on, durability=schema_and_data);

create table dbo.WebRequestParams_Memory
(
    RequestParamId int not null identity(1,1)
        primary key nonclustered
        hash with (bucket_count=5000000),
    RequestId int not null,
    ParamName varchar(64) not null,
    ParamValue nvarchar(256) not null,

    index IDX_RequestID nonclustered hash(RequestID)
    with (bucket_count=1000000)
)
with (memory_optimized=on, durability=schema_and_data);
```

Each memory-optimized table has a DURABILITY setting. The default SCHEMA_AND_DATA value indicates that the data in the tables is fully durable and persists on disk for recovery purposes. Operations on such tables are logged in the database transaction log.

SCHEMA_ONLY is another value, which indicates that data in memory-optimized tables is not durable and would be lost in the event of a SQL Server restart, crash, or failover to another node. Operations against non-durable, memory-optimized tables are not logged in the transaction log. Non-durable tables are extremely fast and can be used if you need to store temporary data in use cases similar to temporary tables in tempdb.

Indexes of memory-optimized tables must be created inline and defined as part of a CREATE TABLE statement. Unfortunately, it is impossible to alter a memory-optimized table and/or create any additional indexes after a table is created.

■ **Tip** You can drop and recreate a memory-optimized table to change its definition and/or indexes.

Memory-optimized tables have other limitations besides the inability to alter them. To name just a few, they cannot have triggers, cannot reference or be referenced with foreign key constraints, nor can they have unique constraints defined. Most importantly, memory-optimized tables do not support off-row (ROW-OVERFLOW and LOB) data storage, which limits the maximum row size to 8,060 bytes and prevents you from using certain data types.

■ **Note** I will discuss memory-optimized tables and their limitations in detail in Chapter 3.

Memory-optimized tables support two types of indexes, HASH and NONCLUSTERED. Hash indexes are optimized for point lookup operations, which is the search of one or multiple rows with equality predicate(s). This is a conceptually new index type in SQL Server, and the Storage Engine does not have anything similar to it implemented. Nonclustered indexes, on the other hand, are somewhat similar to B-Tree indexes on on-disk tables. Note that Microsoft used to call nonclustered indexes *range indexes* in SQL Server 2014 CTP releases and whitepapers.

■ **Note** I will discuss hash indexes in detail in Chapter 4. Nonclustered indexes are covered in Chapter 5.

In-Memory OLTP has one important requirement regarding text columns that participate in the indexes. Those columns must use a binary BIN2 collation. That collation is case- and accent-sensitive, which could be the breaking change in the system behavior when you migrate on-disk tables to become memory-optimized.

■ **Note** I will talk about utilizing In-Memory OLTP in existing systems in Chapter 11.

Working with Memory-Optimized Tables

You can access data in memory-optimized tables either using interpreted T-SQL or from natively compiled stored procedures. In interpreted mode, SQL Server treats memory-optimized tables pretty much the same way as on-disk tables. It optimizes queries and caches execution plans, regardless of where the table is located. The same set of operators is used during query execution. From a high level, when SQL Server needs to get a row from a table, and the operator's `GetRow()` method is called, it is routed either to the Storage Engine or to the In-Memory OLTP Engine, depending on the underlying table type.

Most T-SQL features and constructs are supported in interpreted mode. Some limitations still exist; for example, you cannot truncate a memory-optimized table nor use it as the target in `MERGE` statement. Fortunately, the list of such limitations is very small.

Listing 2-3 shows an example of a T-SQL stored procedure that inserts data into the memory-optimized tables created in Listing 2-2. For simplicity sake, the procedure accepts the data that needs to be inserted into the `dbo.WebRequestParams_Memory` table as the regular parameters limiting it to five values. Obviously, in production code it is better to use table-valued parameters in such a scenario.

Listing 2-3. Stored Procedure That Inserts Data to Memory-Optimized Tables Through the Interop Engine

```
create proc dbo.InsertRequestInfo_Memory
(
   @URL varchar(255)
   ,@RequestType tinyint
   ,@ClientIP varchar(15)
   ,@BytesReceived int
   -- Header fields
   ,@Authorization varchar(256)
   ,@UserAgent varchar(256)
   ,@Host varchar(256)
   ,@Connection varchar(256)
   ,@Referer varchar(256)
   -- Hardcoded parameters.. Just for the demo purposes
   ,@Param1 varchar(64) = null
   ,@Param1Value nvarchar(256) = null
   ,@Param2 varchar(64) = null
   ,@Param2Value nvarchar(256) = null
   ,@Param3 varchar(64) = null
   ,@Param3Value nvarchar(256) = null
   ,@Param4 varchar(64) = null
   ,@Param4Value nvarchar(256) = null
   ,@Param5 varchar(64) = null
   ,@Param5Value nvarchar(256) = null
)
```

```
as
begin
   set nocount on
   set xact_abort on

   declare
      @RequestId int

   begin tran
      insert into dbo.WebRequests_Memory
         (URL,RequestType,ClientIP,BytesReceived)
      values
         (@URL,@RequestType,@ClientIP,@BytesReceived);

      select @RequestId = SCOPE_IDENTITY();

      insert into dbo.WebRequestHeaders_Memory
         (RequestId,HeaderName,HeaderValue)
      values
         (@RequestId,'AUTHORIZATION',@Authorization)
         ,(@RequestId,'USERAGENT',@UserAgent)
         ,(@RequestId,'HOST',@Host)
         ,(@RequestId,'CONNECTION',@Connection)
         ,(@RequestId,'REFERER',@Referer);

      ;with Params(ParamName, ParamValue)
      as
      (
         select ParamName, ParamValue
         from (
            values
              (@Param1, @Param1Value)
               ,(@Param2, @Param2Value)
               ,(@Param3, @Param3Value)
               ,(@Param4, @Param4Value)
               ,(@Param5, @Param5Value)
            ) v(ParamName, ParamValue)
         where
            ParamName is not null and
            ParamValue is not null
      )
      insert into dbo.WebRequestParams_Memory
            (RequestID,ParamName,ParamValue)
         select @RequestID, ParamName, ParamValue
         from Params;
   commit
end
```

As you see, the stored procedure that works through the interop engine does not require any specific language constructs to access memory-optimized tables.

Natively compiled stored procedures are also defined with a regular CREATE PROCEDURE statement and they use T-SQL language. However, there are several additional options that must be specified at the creation stage.

The code in Listing 2-4 creates the natively compiled stored procedure that accomplishes the same logic as the dbo.InsertRequestInfo_Memory stored procedure defined in Listing 2-3.

Listing 2-4. Natively Complied Stored Procedure

```
create proc dbo.InsertRequestInfo_NativelyCompiled
(
   @URL varchar(255) not null
   ,@RequestType tinyint not null
   ,@ClientIP varchar(15) not null
   ,@BytesReceived int not null
   -- Header fields
   ,@Authorization varchar(256) not null
   ,@UserAgent varchar(256) not null
   ,@Host varchar(256) not null
   ,@Connection varchar(256) not null
   ,@Referer varchar(256) not null
   -- Parameters.. Just for the demo purposes
   ,@Param1 varchar(64) = null
   ,@Param1Value nvarchar(256) = null
   ,@Param2 varchar(64) = null
   ,@Param2Value nvarchar(256) = null
   ,@Param3 varchar(64) = null
   ,@Param3Value nvarchar(256) = null
   ,@Param4 varchar(64) = null
   ,@Param4Value nvarchar(256) = null
   ,@Param5 varchar(64) = null
   ,@Param5Value nvarchar(256) = null
)
with native_compilation, schemabinding, execute as owner
as
begin atomic with
(
   transaction isolation level = snapshot
   ,language = N'English'
)
   declare
      @RequestId int

   insert into dbo.WebRequests_Memory
      (URL,RequestType,ClientIP,BytesReceived)
   values
      (@URL,@RequestType,@ClientIP,@BytesReceived);
```

```
    select @RequestId = SCOPE_IDENTITY();

    insert into dbo.WebRequestHeaders_Memory
       (RequestId,HeaderName,HeaderValue)
    values
       (@RequestId,'AUTHORIZATION',@Authorization);

    insert into dbo.WebRequestHeaders_Memory
       (RequestId,HeaderName,HeaderValue)
    values
       (@RequestId,'USERAGENT',@UserAgent);

    insert into dbo.WebRequestHeaders_Memory
       (RequestId,HeaderName,HeaderValue)
    values
       (@RequestId,'HOST',@Host);

    insert into dbo.WebRequestHeaders_Memory
       (RequestId,HeaderName,HeaderValue)
    values
       (@RequestId,'CONNECTION',@Connection);

    insert into dbo.WebRequestHeaders_Memory
       (RequestId,HeaderName,HeaderValue)
    values
       (@RequestId,'REFERER',@Referer);

    if @Param1 collate Latin1_General_100_BIN2
            is not null and
       @Param1Value
          collate Latin1_General_100_BIN2
             is not null
    begin
       insert into dbo.WebRequestParams_Memory
          (RequestID,ParamName,ParamValue)
       values
          (@RequestId,@Param1,@Param1Value);

       if @Param2
             collate Latin1_General_100_BIN2
                is not null and
          @Param2Value
             collate Latin1_General_100_BIN2
                is not null
       begin
          insert into dbo.WebRequestParams_Memory
             (RequestID,ParamName,ParamValue)
          values
             (@RequestId,@Param2,@Param2Value);
```

```
        if @Param3
              collate Latin1_General_100_BIN2
                 is not null and
           @Param3Value
              collate Latin1_General_100_BIN2
                 is not null
        begin
           insert into dbo.WebRequestParams_Memory
              (RequestID,ParamName,ParamValue)
           values
              (@RequestId,@Param3,@Param3Value);

           if @Param4
                 collate Latin1_General_100_BIN2
                    is not null and
              @Param4Value
                 collate Latin1_General_100_BIN2
                    is not null
           begin
              insert into dbo.WebRequestParams_Memory
                 (RequestID,ParamName,ParamValue)
              values
                 (@RequestId,@Param4,@Param4Value);

              if @Param5
                    collate Latin1_General_100_BIN2
                       is not null and
                 @Param5Value
                    collate Latin1_General_100_BIN2
                       is not null
                 insert into dbo.WebRequestParams_Memory
                    (RequestID,ParamName,ParamValue)
                 values
                    (@RequestId,@Param5,@Param5Value);
           end
        end
     end
  end
end
```

You should specify that the stored procedure is natively compiled using the `WITH NATIVE_COMPILATION` clause. All natively-compiled stored procedures are schema-bound, and they require you to specify the `SCHEMABINDING` option. Finally, setting the execution context is a requirement. Natively compiled stored procedures do not support the `EXECUTE AS CALLER` security context to avoid expensive permission checks at execution time, and they require you to specify the `EXECUTE AS OWNER`, `EXECUTE AS USER`, or `EXECUTE AS SELF` context in the definition.

Natively compiled stored procedures execute as atomic blocks indicated by the `BEGIN ATOMIC` keyword, which is an *all or nothing* approach. Either all of the statements in the procedure succeed or all of them fail.

When a natively compiled stored procedure is called outside of the context of an active transaction, it starts a new transaction and either commits or rolls it back at the end of the execution.

In cases where a procedure is called in the context of an active transaction, SQL Server creates a savepoint at the beginning of the procedure's execution. In case of an error in the procedure, SQL Server rolls back the transaction to the created savepoint. Based on the severity and type of error, the transaction is either going to be able to continue and commit or become doomed and uncommittable.

Even though the `dbo.InsertRequestInfo_Memory` and `dbo.InsertRequestInfo_NativelyCompiled` stored procedures accomplish the same task, their implementation is slightly different. Natively compiled stored procedures have a very extensive list of limitations and unsupported T-SQL features. In the example above, you can see that neither the `INSERT` statement with multiple `VALUES` nor CTE/Subqueries were supported. Note that string comparison and manipulation logic require `BIN2` collation, which is not the case in interop mode.

■ **Note** I will discuss natively compiled stored procedures, atomic transactions, and supported T-SQL language constructs in greater depth in Chapter 6.

Finally, it is worth mentioning that natively compiled stored procedures can access only memory-optimized tables. It is impossible to query on-disk tables or, as another example, join memory-optimized and on-disk tables together. You have to use interpreted T-SQL and the interop engine for such tasks.

In-Memory OLTP in Action: Resolving Latch Contention

Latches are lightweight synchronization objects, which SQL Server uses to protect the consistency of internal data structures. Multiple sessions (or, in that context, threads) cannot modify the same object simultaneously.

Consider the situation when multiple sessions try to access the same data page in the buffer pool. While it is safe for the multiple sessions/threads to read the data simultaneously, data modifications must be serialized and have exclusive access to the page. If such rule is not enforced, multiple threads could update a different part of the data page at once, overwriting each other's changes and making the data inconsistent, which would lead to page corruption.

Latches help to enforce that rule. The threads that need to read data from the page obtain shared (S) latches, which are compatible with each other. Data modification, on the other hand, requires an exclusive (X) latch, which prevents other readers and writers from accessing the data page.

■ **Note** Even though latches are conceptually very similar to locks, there is a subtle difference between them. Locks enforce *logical* consistency of the data. For example, they reduce or prevent concurrency phenomena, such as dirty or phantom reads. Latches, on the other hand, enforce *physical* data consistency, such as preventing corruption of the data page structures.

Usually, latches have a very short lifetime and are barely noticeable in the system. However, in very busy OLTP systems, with a large number of CPUs and a high rate of simultaneous data modifications, latch contention can become the bottleneck. You can see the sign of such a bottleneck by the large percent of PAGELATCH waits in wait statistics or by analyzing the `sys.dm_os_latch_stats` data management view.

■ **Tip** One of the common cases of latch contention, allocation maps contention, also presents itself with PAGELATCH waits in wait statistics. The widely known example is PAGELATCH, which indicates contention on the allocation map pages in `tempdb`. You can address such contention by increasing the number of data files in the filegroups with volatile data.

In-Memory OLTP can be extremely helpful in addressing latch contention due to its latch-free architecture. It can help to dramatically increase data modification throughput in some scenarios. In this section, you will see one such example.

In my test environment, I use SQL Server 2014 RTM CU5 installed in the virtual machine with 32 vCPUs and 128GB of RAM. The disk subsystem consists of two separate RAID-10 arrays utilizing 15K SAS drives.

I created the database shown in Listing 2-1 with 16 data files in LOGDATA filegroup in order to minimize allocation maps latch contention. The log file has been placed on one of the disk arrays, while data and In-Memory OLTP filegroups share the second one. It is worth noting that placing on-disk and In-Memory filegroups on the different arrays in production often leads to better I/O performance. However, it will not affect the test scenarios where we do not mix on-disk and In-Memory OLTP workloads in the same tests.

As the first step, let's create the set of on-disk tables that mimics the structure of memory-optimized tables created earlier in the chapter, and the stored procedure that inserts data into those tables. Listing 2-5 shows the code to accomplish this.

Listing 2-5. Creating On-Disk Tables and a Stored Procedure

```
create table dbo.WebRequests_Disk
(
   RequestId int not null identity(1,1),
   RequestTime datetime2(4) not null
      constraint DEF_WebRequests_Disk_RequestTime
      default sysutcdatetime(),
```

```
   URL varchar(255) not null,
   RequestType tinyint not null, -- GET/POST/PUT
   ClientIP varchar(15) not null,
   BytesReceived int not null,

   constraint PK_WebRequests_Disk
   primary key nonclustered(RequestID)
   on [LOGDATA]
) on [LOGDATA];

create unique clustered index IDX_WebRequests_Disk_RequestTime_RequestId
on dbo.WebRequests_Disk(RequestTime,RequestId)
on [LOGDATA];

/* Foreign Keys have not been defined to make
   on-disk and memory-optimized tables as
   similar as possible */
create table dbo.WebRequestHeaders_Disk
(
   RequestId int not null,
   HeaderName varchar(64) not null,
   HeaderValue varchar(256) not null,

   constraint PK_WebRequestHeaders_Disk
   primary key clustered(RequestID,HeaderName)
   on [LOGDATA]
);

create table dbo.WebRequestParams_Disk
(
   RequestId int not null,
   ParamName varchar(64) not null,
   ParamValue nvarchar(256) not null,

   constraint PK_WebRequestParams_Disk
   primary key clustered(RequestID,ParamName)
   on [LOGDATA]
);
go
```

```
create proc dbo.InsertRequestInfo_Disk
(
   @URL varchar(255)
   ,@RequestType tinyint
   ,@ClientIP varchar(15)
   ,@BytesReceived int
   -- Header fields
   ,@Authorization varchar(256)
   ,@UserAgent varchar(256)
   ,@Host varchar(256)
   ,@Connection varchar(256)
   ,@Referer varchar(256)
   -- Parameters.. Just for the demo purposes
   ,@Param1 varchar(64) = null
   ,@Param1Value nvarchar(256) = null
   ,@Param2 varchar(64) = null
   ,@Param2Value nvarchar(256) = null
   ,@Param3 varchar(64) = null
   ,@Param3Value nvarchar(256) = null
   ,@Param4 varchar(64) = null
   ,@Param4Value nvarchar(256) = null
   ,@Param5 varchar(64) = null
   ,@Param5Value nvarchar(256) = null
)
as
begin
   set nocount on
   set xact_abort on

   declare
      @RequestId int

   begin tran
      insert into dbo.WebRequests_Disk
         (URL,RequestType,ClientIP,BytesReceived)
      values
         (@URL,@RequestType,@ClientIP,@BytesReceived);

      select @RequestId = SCOPE_IDENTITY();

      insert into dbo.WebRequestHeaders_Disk
         (RequestId,HeaderName,HeaderValue)
      values
         (@RequestId,'AUTHORIZATION',@Authorization)
         ,(@RequestId,'USERAGENT',@UserAgent)
         ,(@RequestId,'HOST',@Host)
         ,(@RequestId,'CONNECTION',@Connection)
         ,(@RequestId,'REFERER',@Referer);
```

```
      ;with Params(ParamName, ParamValue)
      as
      (
         select ParamName, ParamValue
         from (
            values
               (@Param1, @Param1Value)
               ,(@Param2, @Param2Value)
               ,(@Param3, @Param3Value)
               ,(@Param4, @Param4Value)
               ,(@Param5, @Param5Value)
            ) v(ParamName, ParamValue)
         where
            ParamName is not null and
            ParamValue is not null
      )
      insert into dbo.WebRequestParams_Disk
         (RequestID,ParamName,ParamValue)
         select @RequestId, ParamName, ParamValue
            from Params;
   commit
end;
```

In these tests, we will compare insert throughput of on-disk and memory-optimized tables using `dbo.InsertRequestInfo_Disk`, `dbo.InsertRequestInfo_Memory`, and `dbo.InsertRequestInfo_NativelyCompiled` stored procedures, calling them simultaneously from the multiple sessions in the loop. Each call will insert one row into the `dbo.WebRequests` table, five rows into the `dbo.WebRequestHeaders` table, and from one to five rows into the `dbo.WebRequestDisks` table, which makes nine rows total in average in the single transaction.

■ **Note** The test application and scripts are included in the companion materials of the book.

In case of the `dbo.InsertRequestInfo_Disk` stored procedure and on-disk tables, my test server achieved a maximum throughput of about **3,700–3,800** batches/calls per second with **45** concurrent sessions. A further increase in the number of sessions did not help and, in fact, even slightly reduced the throughput. Figure 2-1 shows several performance counters at time of test.

\\SQL14TEST	
Processor Information	**_Total**
% Processor Time	12.706
SQLServer:Latches	
Average Latch Wait Time (ms)	0.362
Latch Waits/sec	29,149.047
Total Latch Wait Time (ms)	10,539.786
SQLServer:SQL Statistics	
Batch Requests/sec	3,773.417

Figure 2-1. *Performance counters when data was inserted into on-disk tables*

Even though we maxed out insert throughput, CPU load on the server was very low, which clearly indicates that the CPU was not the bottleneck during the test. At the same time, the server suffered from the large number of latches, which were used to serialize access to the data pages in the buffer pool. Even though wait time of each individual latch was very low, the total latch wait time was high due to the excessive number of them acquired every second.

You can confirm that latches were the bottleneck by analyzing wait statistics collected during the test. Figure 2-2 illustrates the output from the `sys.dm_os_wait_stats` view. You can see that latch waits are at the top of the list.

	wait_type	wait_time_ms	waiting_tasks_count	Avg Wait Time (ms)	Percent
1	PAGELATCH_EX	1461382	3231793	0	53.6981
2	WRITELOG	1134132	440517	2	41.6734
3	PAGELATCH_SH	102513	231690	0	3.7668
4	LATCH_SH	10203	17027	0	0.3749

Figure 2-2. *Wait statistics collected during the test (insert into on-disk tables)*

The situation changed when I repeated the tests with the `dbo.InsertRequestInfo_Memory` stored procedure, which inserted data into memory-optimized tables through the interop engine. I maxed out the throughput with **150** concurrent sessions, which is more than three times more sessions compared to the previous test with on-disk tables. In this scenario, SQL Server was able to handle **30,000–32,000** batches/calls per second. A further increase in the number of concurrent sessions did not change the throughput; however, the duration of each call linearly increased as more sessions were added.

Figure 2-3 illustrates performance counters during the test. As you see, there were no latches with memory-optimized tables and the CPUs were fully utilized.

\\SQL14TEST	
Processor Information	**_Total**
% Processor Time	87.993
SQLServer:Latches	
Average Latch Wait Time (ms)	0.000
Latch Waits/sec	0.000
Total Latch Wait Time (ms)	0.000
SQLServer:SQL Statistics	
Batch Requests/sec	31,849.486

Figure 2-3. *Performance counters when data was inserted into memory-optimized tables through the interop engine*

As you can see in Figure 2-4, the only significant wait in the system is `WRITELOG`, which is related to the transaction log write performance.

	wait_type	wait_time_ms	waiting_tasks_count	Avg Wait Time (ms)	Percent
1	WRITELOG	12887581	5205636	2	96.8793
2	WAIT_XTP_HOST_WAIT	410496	33	12439	3.0858
3	LOGPOOL_FREEPOOLS	2675	7247	0	0.0201

Figure 2-4. *Wait statistics collected during the test (insert into memory-optimized tables through interop engine)*

The natively compiled `dbo.InsertRequestInfo_NativelyCompiled` stored procedure improved the situation even further. With **150** concurrent sessions, SQL Server was able to handle **45,000-50,000** batches/calls per second, which translates to 400,000-450,000 individual inserts per second.

Figure 2-5 illustrates performance counters during test execution. Even with the increase in throughput, the natively compiled stored procedure put less load on the CPU than the interop engine, and disk performance became the clear bottleneck in this setup.

\\SQL14TEST	
Processor Information	**_Total**
% Processor Time	74.999
SQLServer:Latches	
Average Latch Wait Time (ms)	0.000
Latch Waits/sec	0.000
Total Latch Wait Time (ms)	0.000
SQLServer:SQL Statistics	
Batch Requests/sec	49,553.926

Figure 2-5. *Performance counters when data was inserted into memory-optimized tables using natively compiled stored procedure*

Waits in the wait statistics are very similar to the previous test, with WRITELOG wait at the top of the list (see Figure 2-6).

	wait_type	wait_time_ms	waiting_tasks_count	Avg Wait Time (ms)	Percent
1	WRITELOG	6594381	3901606	1	95.8966
2	WAIT_XTP_HOST_WAIT	280020	23	12174	4.0721

Figure 2-6. *Wait statistics collected during the test (insert into memory-optimized tables using natively compiled stored procedure)*

We can confirm that disk performance has become the limiting factor in our setup by running the same test with non-durable, memory-optimized tables. You can do this by dropping and recreating the database, and creating the same set of memory-optimized tables using the DURABILITY=SCHEMA_ONLY option. No other code changes are required.

Figure 2-7 shows performance counters collected during the test with **225** concurrent sessions calling the dbo.InsertRequestInfo_NativelyCompiled stored procedure to insert data into non-durable tables. As you can see, in that scenario we were able to fully utilize the CPU on the system after we removed the I/O bottleneck, which improve throughput for another 50% compared to durable memory-optimized tables.

\\SQL14TEST	
Processor Information	**_Total**
% Processor Time	93.115
SQLServer:Latches	
Average Latch Wait Time (ms)	0.000
Latch Waits/sec	0.000
Total Latch Wait Time (ms)	0.000
SQLServer:SQL Statistics	
Batch Requests/sec	78,557.550

Figure 2-7. *Performance counters when data was inserted into non-durable memory-optimized tables using a natively compiled stored procedure*

Finally, it is worth noting that In-Memory OLTP uses different and more efficient logging, which leads to a much smaller transaction log footprint. Figure 2-8 illustrates log file write statistics collected during 1 minute of test execution using sys.dm_io_virtual_file_stats DMF. The order of outputs in the figure corresponds to the order in which the tests were run: on-disk table inserts, inserts into memory-optimized tables through the interop engine, and natively compiled stored procedures, respectively.

	File Name	Written MB	Writes	IO Count	Write Stall	Avg Write Stall
3	InMemoryOLTPDemo_log	721.027	13624	13624	11670	0.857

	File Name	Written MB	Writes	IO Count	Write Stall	Avg Write Stall
2	InMemoryOLTPDemo_log	2132.637	144648	144651	237337	1.641

	File Name	Written MB	Writes	IO Count	Write Stall	Avg Write Stall
2	InMemoryOLTPDemo_log	2451.434	125636	125642	179405	1.428

Figure 2-8. *Transaction log write statistics during the tests*

As you see, in interop mode In-Memory OLTP inserted about eight times more data; however, it used just three times more space in the transaction log than with on-disk tables. The situation is even better with natively compiled stored procedures. Even though it wrote about 15 percent more to the log, it inserted about 50 percent more data compared to interop mode.

■ **Note** I will discuss In-Memory OLTP transaction logging in greater depth in Chapter 8.

Obviously, different scenarios will lead to different results, and performance improvements would greatly depend on the hardware, database schema, and use-case and workload in the system. However, it is not uncommon to see 3x-5x improvements when you access memory-optimized tables through the interop engine and a 10x-30x rate with natively compiled stored procedures.

More importantly, In-Memory OLTP allows us to improve the performance of the system by scaling up and upgrading hardware. For example, in our scenario we can achieve better throughput by adding more CPUs and/or increasing I/O performance. This would be impossible to do with on-disk tables where latch contention becomes the bottleneck.

Summary

The In-Memory OLTP Engine is fully integrated into the Enterprise Edition of SQL Server and is always installed with the 64-bit version of the product. However, every database that uses In-Memory OLTP objects should have the separate In-Memory OLTP filegroup created. This filegroup should be placed in the disk array optimized for sequential I/O performance.

You can create memory-optimized tables with the regular `CREATE TABLE` statement marking tables as `MEMORY_OPTIMIZED` and specifying table durability option. The data in the tables with `SCHEMA_AND_DATA` durability is persisted on disk. Tables with `SCHEMA_ONLY` durability do not persist the data and they can be used as In-Memory temporary tables that provide extremely fast performance.

Memory-optimized tables do not support all T-SQL features and data types. Moreover, they cannot have rows that exceed 8,060 bytes in size and they do not support `ROW-OVERFLOW` and `LOB` storage. Finally, memory-optimized tables and indexes cannot be altered after the table was created.

You can access memory-optimized tables from either interpreted T-SQL through the interop engine or from natively compiled stored procedures. Almost all T-SQL features are supported in interpreted mode. Conversely, natively compiled stored procedures support a very limited set of features; however, they can introduce significant performance improvements compared to the interop engine.

CHAPTER 3

Memory-Optimized Tables

This chapter discusses memory-optimized tables in detail. It shows how memory-optimized tables store their data and how SQL Server accesses them. It covers the format of the data rows in memory-optimized tables and talks about the process of native compilation.

Finally, the chapter provides an overview of the memory-optimized tables limitations that exist in the first release of the In-Memory OLTP Engine.

On-Disk vs. Memory-Optimized Tables

Data and index structures in memory-optimized tables are different from those in on-disk tables. In on-disk tables, the data is stored in the 8KB data pages grouped together in eight-page extents on per-index or per-heap basis. Every page stores the data from one or multiple data rows. Moreover, the data from variable-length or LOB columns can be stored off-row on `ROW_OVERFLOW` and `LOB` data pages when it does not fit on one in-row page.

All pages and rows in on-disk tables are referenced by in-file offsets, which is the combination of `file_id`, data page offset/position in the file and, in case of a data row, row offset/position on the data page.

Finally, every nonclustered index stores its own copy of the data from the index key columns referencing the main row by *row-id*, which is either the clustered index key value or a physical address (offset) of the row in the heap table.

Figures 3-1 and 3-2 illustrate these concepts. They show clustered and nonclustered index B-Trees defined on a table. As you see, pages are linked through in-file offsets. The nonclustered index persists the separate copy of the data and references the clustered index through clustered index key values.

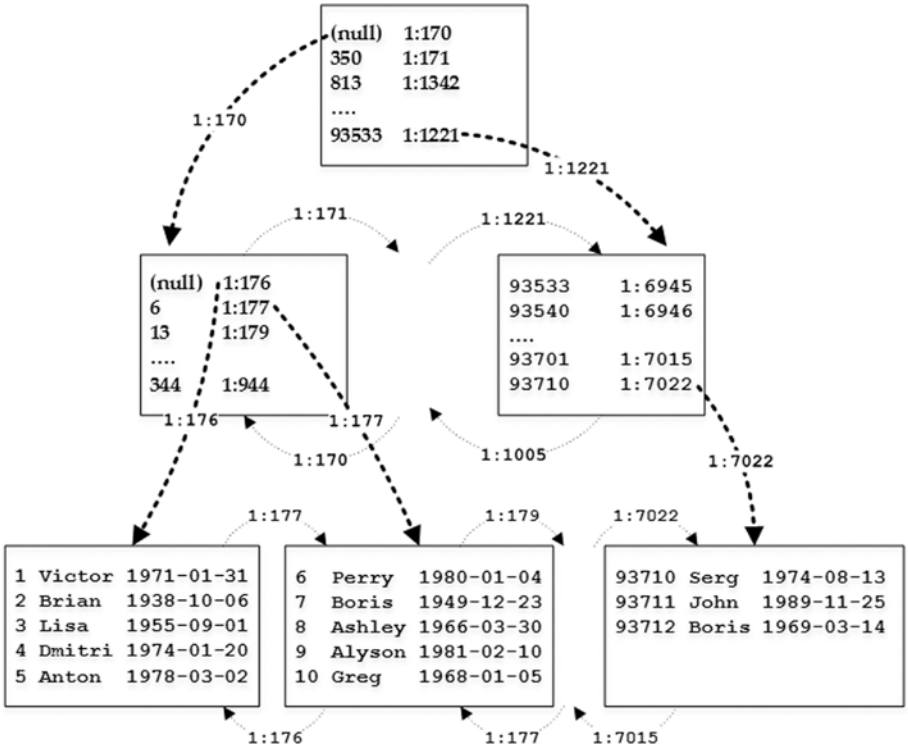


Figure 3-1. *Clustered index on on-disk tables*

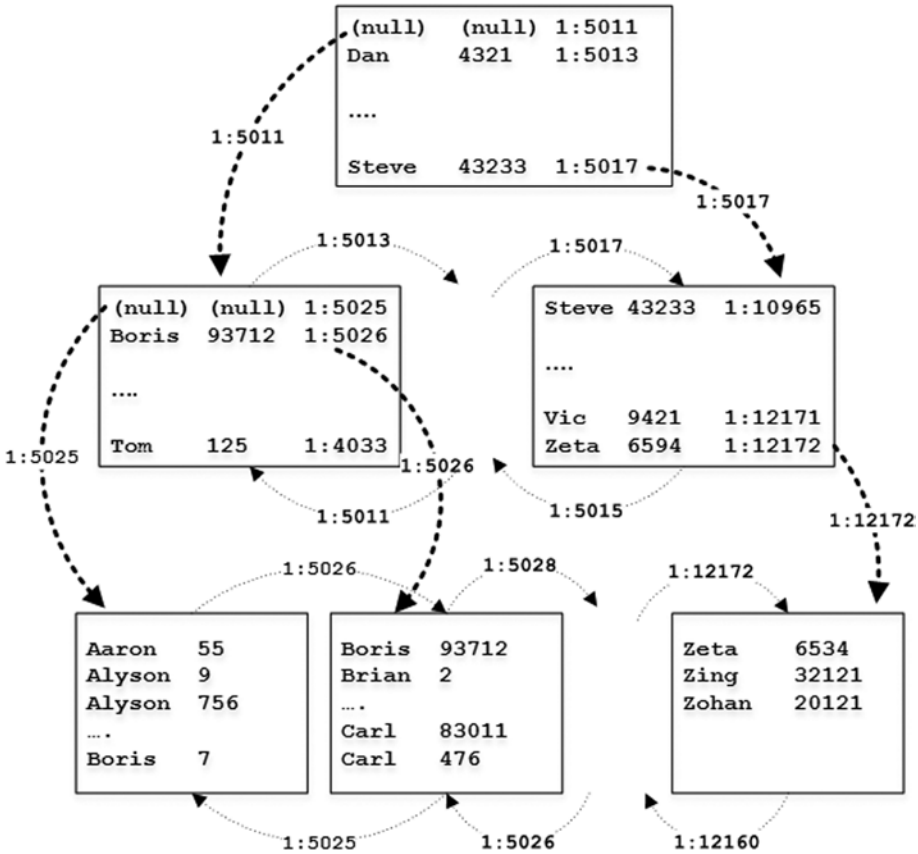


Figure 3-2. *Nonclustered index on on-disk tables*

Every time you need to access the data from the page, SQL Server loads the copy of the page to the memory, caching it in the buffer pool. However, the format and structure of the data page in the buffer pool does not change, and pages there still use in-file offsets to reference each other. The SQL Server component called the *Buffer Manager* manages the buffer pool, and it tracks the data page's in-memory locations, translating in-file offsets to the corresponding memory addresses of the page structures.

Consider the situation when SQL Server needs to scan several data pages in the index. The Scheduler requests the page from the Buffer Manager, using `file_id` and `page_id` to identify it. The Buffer Manager, in turn, checks if the page is already cached, reading it from disk when necessary. When the page is read and processed, SQL Server obtains the address of the next page in the index and repeats the process.

It is also entirely possible that SQL Server needs to access multiple pages in order to read a single row. This happens in case of off-row storage and/or when the execution plan uses nonclustered indexes and issues *Key* or *RID Lookup* operations, obtaining the data from the clustered index or heap.

The process of locating a page in the buffer pool is very fast; however, it still introduces overhead that affects performance of the queries. The performance hit is much worse when the data page is not in memory and a physical I/O operation is required.

The In-Memory OLTP Engine uses a completely different approach with memory-optimized tables. With the exception of Bw-Trees in nonclustered indexes, which I will discuss in Chapter 5, in-memory objects do not use data pages. Data rows reference each other through the memory pointers. Every row knows the memory address of a next row in the chain, and SQL Server does not need to do any extra steps to locate it.

Every memory-optimized table has at least one index row chain to link rows together and, therefore, every table must have at least one index defined. In the case of durable memory-optimized tables, there is the requirement of creating a primary key constraint, which can serve for such a purpose.

To illustrate the concepts of row chains, let's create the memory-optimized table as shown in Listing 3-1.

Listing 3-1. Creating the Memory-Optimized Table

```
create table dbo.People
(
   Name varchar(64)
         collate Latin1_General_100_BIN2 not null
      constraint PK_People
      primary key nonclustered
      hash with (bucket_count = 1024),
   City varchar(64)
         collate Latin1_General_100_BIN2 not null,

   index IDX_City
   nonclustered hash(City)
   with (bucket_count = 1024),
)
with (memory_optimized = on, durability = schema_only);
```

This table has two hash indexes defined on the `Name` and `City` columns. I am not going to discuss hash indexes in depth here but as a general overview, they consist of a hash table (an array of hash buckets), each of which contains a memory pointer to the data row. SQL Server applies a hash function to the index key columns, and the result of the function determines to which bucket a row belongs. All rows that have the same hash value and belong to the same bucket are linked together in a row chain; every row has a pointer to the next row in a chain.

■ **Note** I will discuss hash indexes in detail in Chapter 4.

Figure 3-3 illustrates this. Solid arrows represent pointers in the index on the `Name` column. Dotted arrows represent pointers in the index on the `City` column. For simplicity sake, let's assume that the hash function generates a hash value based on the first letter of the string. Two numbers, displayed in each row, indicate row lifetime, which I will explain in the next section of this chapter.

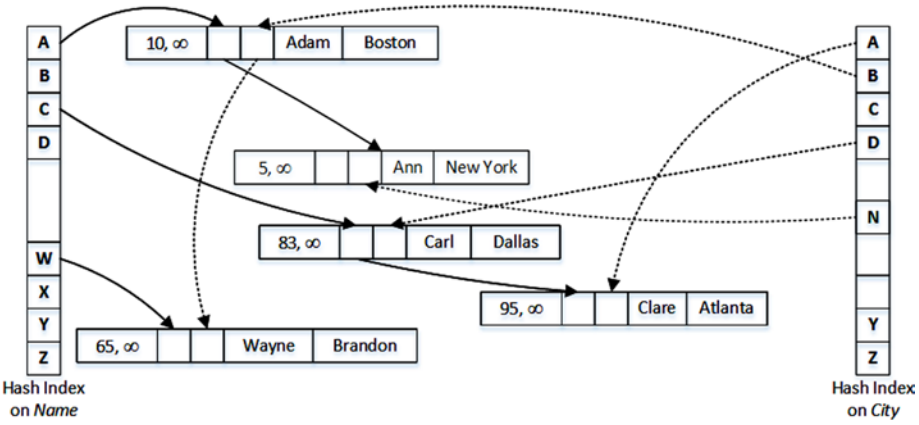


Figure 3-3. *Memory-optimized table with two hash indexes*

In contrast to on-disk tables, indexes on memory-optimized tables are not created as separate data structures but rather *embedded* as pointers in the data rows, which, in a nutshell, makes every index *covering*.

■ **Note** To be precise, nonclustered indexes on memory-optimized tables introduce additional data structures in memory. However, they are much more efficient compared to nonclustered indexes on on-disk tables and do not require *Key* or *RID Lookup* operations to access the data. I will discuss nonclustered indexes in details in Chapter 5.

Introduction to the Multiversion Concurrency Control

As you already noticed in Figure 3-3, every row in a memory-optimized table has two values, called `BeginTs` and `EndTs`, which define the lifetime of the row. A SQL Server instance maintains the *Global Transaction Timestamp* value, which is auto-incremented when the transaction commits and is unique for every committed transaction. `BeginTs` stores the Global Transaction Timestamp of the transaction that is inserted a row, and `EndTs` stores the timestamp of the transaction that deleted a row. A special value called `Infinity` is used as `EndTs` for the rows that have not been deleted.

The rows in memory-optimized tables are never updated. The update operation creates the new version of the row with the new Global Transaction Timestamp set as `BeginTs` and marks the old version of the row as deleted by populating the `EndTs` timestamp with the same value.

Every transaction has a *transaction timestamp*, which is the Global Transaction Timestamp value at the moment the transaction starts. `BeginTs` and `EndTs` control the visibility of a row for the transactions. A transaction can see a row only when its transaction timestamp is between the `BeginTs` and `EndTs` timestamps of the row.

To illustrate that, let's assume that we ran the statement shown in Listing 3-2 and committed the transaction when the Global Transaction Timestamp value was 100.

Listing 3-2. Updating Data in the dbo.People Table

```
update dbo.People
set City = 'Cincinnati'
where Name = 'Ann'
```

Figure 3-4 illustrates the data in the table after an update transaction has been committed. As you see, we now have two rows with `Name='Ann'` and different lifetime. The new row has been appended to the row chain referenced by the hash backet for the value of `'A'` in the index on the `Name` column. The hash index on `City` column did not have any rows referenced by the `'C'` bucket, therefore the new row becomes the first in the row chain referenced from that bucket.

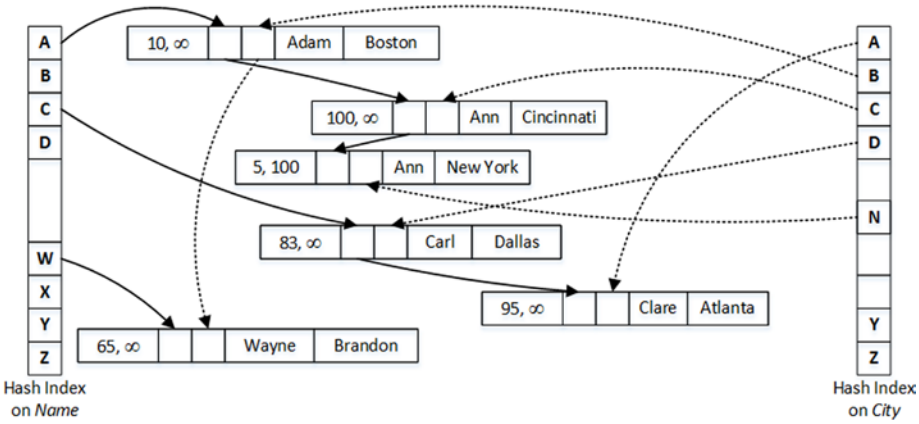


Figure 3-4. *Data in the table after update*

Let's assume that you need to run a query that selects all rows with `Name='Ann'` in the transaction, which started when the Global Transaction Timestamp was 110. SQL Server calculates the hash value for Ann, which is *'A'*, and finds the corresponding bucket in the hash index on the `Name` column. It follows the pointer from that bucket, which references a row with `Name='Adam'`. This row has `BeginTs` of 10 and `EndTs` of `Infinity`; therefore, it is visible to the transaction. However, the `Name` value does not match the predicate and the row is ignored.

In the next step, SQL Server follows the pointer from the Adam index pointer array, which references the first Ann row. This row has BeginTs of 100 and EndTs of Infinity; therefore, it is visible to the transaction and needs to be selected.

As a final step, SQL Server follows the next pointer in the index. Even though the last row also has Name='Ann', it has EndTs of 100 and is invisible to the transaction.

As you should have already noticed, this concurrency behavior and data consistency corresponds to the SNAPSHOT transaction isolation level when every transaction *sees* the data as of the time transaction started. SNAPSHOT is default transaction isolation level in the In-Memory OLTP Engine, which also supports REPEATABLE READ and SERIALIZABLE isolation levels. However, REPEATABLE READ and SERIALIZABLE transactions in In-Memory OLTP behave differently than with on-disk tables. In-Memory OLTP raises an exception and rolls back a transaction if REPEATABLE READ or SERIALIZABLE data consistency rules were violated rather than blocks a transacton as with on-disk tables.

In-Memory OLTP documentation also indicates that autocommitted (single statement) transactions can run in READ COMMITTED isolation level. However, this is a bit misleading. SQL Server promotes and executes such transactions in the SNAPSHOT isolation level and does not require you to explicitly specify the isolation level in your code. The Autocommitted READ COMMITTED transaction would not *see* the changes committed after the transaction started, which is a different behavior compared to the READ COMMITTED transactions against on-disk tables.

■ **Note** I will discuss concurrency model in In-Memory OLTP in Chapter 7.

SQL Server keeps track of the active transactions in the system and detects stale rows with the EndTs timestamp older than the Global Transaction Timestamp of the *oldest active transaction* in the system. Stale rows are invisible for active transactions in the system, and eventually they are removed from the index row chains and deallocated by the garbage collection process.

■ **Note** The garbage collection process is covered in more detail in Chapter 9.

Data Row Format

As you can guess, the format of the data rows in memory-optimized tables is entirely different from on-disk tables and consists of two different sections, *Row Header* and *Payload,* as shown in Figure 3-5.

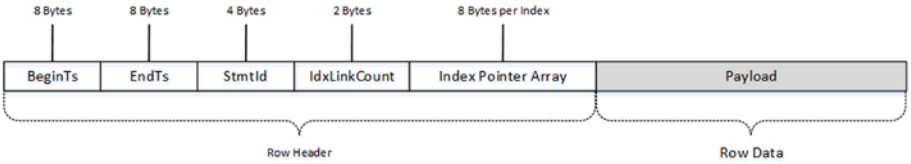


Figure 3-5. *The structure of a data row in a memory-optimized table*

You are already familiar with the `BeginTs` and `EndTs` timestamps in the row header. The next element there is `StmtId`, which references the statement that is inserted that row. Every statement in a transaction has a unique 4-byte `StmtId` value, which works as a *Halloween protection* technique and allows the statement to skip rows it just inserted.

HALLOWEEN PROTECTION

The Halloween effect is a known problem in the relation databases world. It was discovered by IBM researchers almost 40 years ago. In a nutshell, it refers to the situation when the execution of a data modification query is affected by the previous modifications it performed.

You can think of the following statement as a classic example of the Halloween problem:

```
insert into T
      select * from T
```

Without Halloween protection, this query would fall into an infinitive loop, reading the data it just inserted, and inserting it over and over again.

With on-disk tables, SQL Server implements Halloween protection by adding *spool* operators to the execution plan. These operators create a temporary copy of the data before processing it. In our example, all data from the table is cached in the *Table Spool* first, which will work as the source of the data for the insert.

`StmtId` helps to avoid the Halloween problem in memory-optimized tables. Statements check the `StmtId` of the rows, and skip those they just inserted.

The next element in the header, the 2-byte `IdxLinkCount`, indicates how many indexes (pointers) reference the row (or, in the other words, in how many index chains this row is participating). SQL Server uses it to detect rows that can be deallocated by the garbage collection process.

An array of 8-byte index pointers is the last element of the row header. As you already know, every memory-optimized table should have at least one index to link data rows together. At most, you can define eight indexes per memory-optimized table, including the primary key constraint.

The actual row data is stored in the Payload section of the row. The Payload format may vary depending on the table schema. SQL Server works with the Payload through a DLL that is generated and compiled for the table (more on that in the next section of this chapter).

I would like to reiterate that a key principle of In-Memory OLTP is that Payload data is never updated. When a table row needs to be updated, In-Memory OLTP *deletes* the version of the row by setting the `EndTs` attribute of the original row and inserts the new data row version with the new `BeginTs` value and an `EndTs` value of `Infinity`.

Native Compilation of Memory-Optimized Tables

One of the key differences between the Storage and In-Memory OLTP Engines resides in how engines work with row data. The data in on-disk tables is always stored using the same and pre-defined data row format. Strictly speaking, there are several different storage formats based on data compression settings and type of rows; however, the number of possible formats are very small and they do not depend on the table schema. For example, clustered indexes from the multiple tables defined with the same data compression option would store the data in the same way regardless of the tables' schemas.

As usual, that approach comes with benefits and downsides. It is extremely flexible and allows us to alter a table and mix per- and post-altered versions of the rows together. For example, adding a new nullable column to the table is the metadata-level operation, which does not change existing rows. The Storage Engine analyzes table metadata and different row attributes, and handles multiple versions of the rows correctly.

However, such flexibility comes at cost. Consider the situation when the query needs to access the data from the variable-length column in the row. In this scenario, SQL Server needs to find the offset of the variable-length array section in the row, calculate an offset and length of the column data from that array, and analyze if the column data is stored in-row or off-row before getting the required data. All of that can lead to thousands of CPU instructions to execute.

The In-Memory OLTP Engine uses a completely opposite approach. SQL Server creates and compiles the separate DLLs for every memory-optimized table in the system. Those DLLs are loaded into the SQL Server process, and they are responsible for accessing and manipulating the data in Payload section of the row. The In-Memory OLTP Engine is generic and it does not know anything about underlying row structures; all data access is done through those DLLs.

As you can guess, this approach significantly reduces processing overhead; however, it comes at the cost of reduced flexibility. In the first release of In-Memory OLTP, generated DLLs require all rows to have the same structure and, therefore, it is impossible to alter the table after it is created.

This restriction can lead to supportability and performance issues when tables and indexes are defined incorrectly. One such example is the wrong hash index bucket count estimation, which can lead to an excessive number of rows in the row chains, which reduces index seek performance. I will discuss this problem in detail in Chapter 4.

■ **Note** SQL Server places the source code and compiled dlls in the `XTP` subfolder of the SQL Server `DATA` directory. I will talk about those files and the native compilation process in more detail in Chapter 6.

Memory-Optimized Tables: Surface Area and Limitations

The first release of the In-Memory OLTP Engine has an extensive list of limitations. Let's look at those limitations in detail.

Supported Data Types

As mentioned, memory-optimized tables do not support off-row storage and do restrict the maximum data row size to 8,060 bytes. Therefore, only a subset of the data types is supported. The supported list includes:

- `bit`
- Integer types: `tinyint`, `smallint`, `int`, `bigint`
- Floating and fixed point types: `float`, `real`
- `numeric`, and `decimal`. The In-Memory OLTP Engine uses either 8 or 16 bytes to store such data types, which is different from on-disk tables where storage size can be 5, 9, 13, or 17 bytes, depending on precision.
- Money types: `money` and `smallmoney`
- Date/time types: `smalldatetime`, `datetime`, `datetime2`, `date`, and `time`. The In-Memory OLTP Engine uses 4 bytes to store values of `date` data type and 8 bytes for the other data types, which is different from on-disk tables where storage size is based on precision.
- `uniqueidentifiers`
- Non-LOB string types: `(n)char(N)`, `(n)varchar(N)`, and `sysname`
- Non-LOB binary types: `binary(N)` and `varbinary(N)`

Unfortunately, you cannot use any data types that use LOB storage. None of the following data types are supported: `(n)varchar(max)`, `xml`, `clr` data types, `(n)text`, and `image`.

It is also worth remembering that the maximum row size limitation of 8,060 bytes applies to the size of the columns in table definition rather than to the actual row size. For example, it is impossible to define memory-optimized tables with two `varchar(4100)` columns even if you plan to keep data row sizes below the 8,060 bytes threshold.

Constraints and Table Features

In addition to the limited set of supported data types and inability to alter the table, memory-optimized tables have other requirements and limitations. None of the following objects are supported:

- `FOREIGN KEY` constraints
- `CHECK` constraints
- `UNIQUE` constraints or indexes with exception of the `PRIMARY KEY`
- `DML` triggers
- `IDENTITY` columns with `SEED` and `INCREMENT` different than (1,1)
- Computed and sparse columns
- Non-binary collations for the text columns participating in the indexes
- Nullable indexed columns. A column can be defined as nullable when it does not participate in the indexes.

Every memory-optimized table, durable or non-durable, should have at least one and at most eight indexes. Moreover, the durable memory-optimized table should have a unique primary key constraint defined. This constraint is counted as one of the indexes towards the eight-index limit.

If is also worth noting that columns participating in the primary key constraint are non-updatable. You can delete the old and insert the new row as the workaround.

Database-Level Limitations

In-Memory OLTP has several limitations that affect some of the database settings and operations. They include the following:

- You cannot create a Database Snapshot on databases that use In-Memory OLTP.
- The `AUTO_CLOSE` database option must be set to `OFF`.
- `CREATE DATABASE FOR ATTACH_REBUILD_LOG` is not supported.
- `DBCC CHECKDB` skips the memory-optimized tables.
- `DBCC CHECKTABLE` fails if called to check memory-optimized table.

■ **Note** You can see the full list of limitations in the first release of the In-Memory OLTP at `http://msdn.microsoft.com/en-us/library/dn246937.aspx`.

High Availability Technologies Support

Memory-optimized tables are fully supported in an AlwaysOn Failover Cluster and Availability Groups, and with Log Shipping. However, in the case of a Failover Cluster, data from durable memory-optimized tables must be loaded into memory in case of a failover, which could increase failover time.

In the case of AlwaysOn Availability Groups, only durable memory-optimized tables are replicated to secondary nodes. You can access and query those tables on the readable secondary nodes if needed. Data from non-durable memory-optimized tables, on the other hand, is not replicated and will be lost in the case of a failover.

You can set up transactional replication on databases with memory-optimized tables; however, those tables cannot be used as articles in publications.

In-Memory OLTP is not supported in database mirroring sessions. This does not appear to be a big limitation, however. In-Memory OLTP is an Enterprise Edition feature, which allows you to replace database mirroring with AlwaysOn Availability Groups.

Summary

As the opposite to on-disk tables, where data is stored in 8KB data pages, memory-optimized tables link data rows into the index row chains using regular memory pointers. Every row has multiple pointers, one per index row chain. Every table must have at least one and at most eight indexes defined.

A SQL Server instance maintains the Global Transaction Timestamp value, which is auto-incremented when the transaction commits and is unique for every committed transaction. Every data row has `BeginTs` and `EndTs` timestamps that define row lifetimes. A transaction can see a row only when its transaction timestamp (timestamp at time when transaction starts) is between the `BeginTs` and `EndTs` timestamps of the row.

Row data in memory-optimized tables are never updated. When a table row needs to be updated, In-Memory OLTP creates the new version of the row with new `BeginTs` value and *deletes* the old version of the row by populating its `EndTs` timestamp.

SQL Server generates and compiles native DLLs for every memory-optimized table in the system. Those DLLs are loaded into the SQL Server process, and they are responsible for accessing and manipulating the row data.

The first release of In-Memory OLTP has an extensive list of limitations. Those limitations include the inability to alter the table after it is created; a 8,060 byte maximum data row size limit without any off-row storage support; the inability to define triggers, foreign key, check, and unique constraints on tables; and quite a few others.

The In-Memory OLTP Engine is fully supported in AlwaysOn Failover Clusters, Availability Groups, and Log Shipping. Databases with memory-optimized tables can participate in transactional replication; however, you cannot replicate memory-optimized tables.

CHAPTER 4

Hash Indexes

This chapter discusses hash indexes, the new type of indexes introduced in the In-Memory OLTP Engine. It will show their internal structure and explain how SQL Server works with them. You will learn about the most critical property of hash indexes, `bucket_count`, which defines the number of hash buckets in the index hash array. You will see how incorrect bucket count estimations affect system performance. Finally, this chapter talks about the SARGability of hash indexes and statistics on memory-optimized tables.

Hashing Overview

Hashing is a widely-known concept in Computer Science that performs the transformation of the data into short fixed-length values. Hashing is often used in scenarios when you need to optimize point-lookup operations that search within the set of large strings or binary data using equality predicate(s). Hashing significantly reduces an index key size, making them compact, which, in turn, improves the performance of lookup operations.

A properly defined hashing algorithm, often called a *hash function*, provides relatively random hash distribution. A hash function is always deterministic, which means that the same input always generates the same hash value. However, a hash function does not guarantee uniqueness, and different input values can generate the same hashes. That situation is called *collision* and the chance of it greatly depends on the quality of the hash algorithm and the range of allowed hash keys. For example, a hash function that generates a 2-byte hash has a significantly higher chance of collision compared to a function that generates a 4-byte hash.

Hash tables, often called *hash maps*, are the data structures that store hash keys, mapping them to the original data. The hash keys are assigned to *buckets*, in which original data can be found. Ideally, each unique hash key is stored in the individual bucket; however, when the number of buckets in the table is not big enough, it is entirely possible that multiple unique hash keys would be placed into the same bucket. Such situation is also often referenced as a *hash collision* in context of hash tables.

■ **Tip** The `HASHBYTES` function allows you to generate hashes in T-SQL using one of the industry standard algorithms such as `MD5`, `SHA2_512`, and a few others. However, the output of the `HASHBYTES` function is not ideal for point-lookup optimization due to the large size of the output. You can use a `CHECKSUM` function that generates a 4-byte hash instead.

You can index the hash generated by the `CHECKSUM` function and use it as the replacement for the indexes on `uniqueidentifier` columns. It is also useful when you need to perform point-lookup operations on the large (>900 bytes) strings or binary data, which cannot be indexed. I discussed this scenario in Chapter 6 of my book *Pro SQL Server Internals*.

Much Ado About Bucket Count

In the In-Memory OLTP Engine, hash indexes are, in a nutshell, hash tables with buckets implemented as array of a predefined size. Each bucket contains a pointer to a data row. SQL Server applies a hash function to the index key values, and the result of the function determines to which bucket a row belongs. All rows that have the same hash value and belong to the same bucket are linked together through a chain of index pointers in the data rows.

Figure 4-1 illustrates an example of a memory-optimized table with two hash indexes defined. You saw this diagram in the previous chapter; it's displayed here for reference purposes. Remember that in this example we assumed that a hash function generates a hash value based on the first letter of the string. Obviously, a real hash function used in In-Memory OLTP is much more random and does not use character-based hashes.

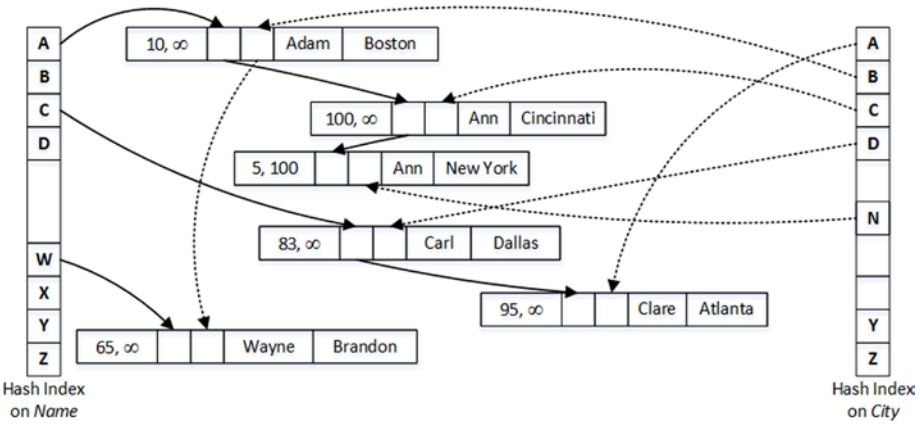


Figure 4-1. *A memory-optimized table with two hash indexes*

The number of buckets is the critical element in hash index performance. An efficient hash function allows you to avoid most collisions during hash key generation; however, you will have collisions in the hash table when the number of buckets is not big enough, and SQL Server has to store different hashes together in the same buckets. Those collisions lead to longer row chains, which requires SQL Server to scan more rows during the query processing.

Bucket Count and Performance

Let's consider a hash function that generates a hash based on the first two letters of the string and can return 26 * 26 = 676 different hash keys. This is a purely hypothetical example, which I am using just for illustration purposes.

Assuming that the hash table can accommodate all 676 different hash buckets and you have the data shown in Figure 4-2, you will need to traverse at most two rows in the chain when you run a query that looks for a specific value.

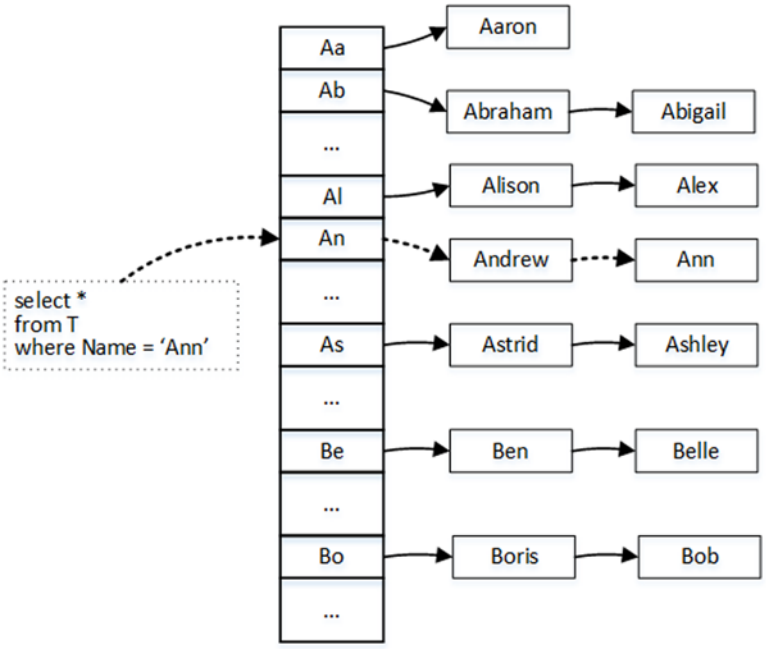


Figure 4-2. *Hash table lookup: 676 buckets*

The dotted arrows in Figure 4-2 illustrate the steps needed to look up the rows for Ann. The process requires you to traverse two rows after you find the right hash bucket in the table.

However, the situation changes if your hash table does not have enough buckets to separate unique hash keys from each other. Figure 4-3 illustrates the situation when a hash table has only 26 buckets and each of them stores multiple different hash keys. Now the same lookup of the Ann row requires you to traverse the chain of nine rows total.

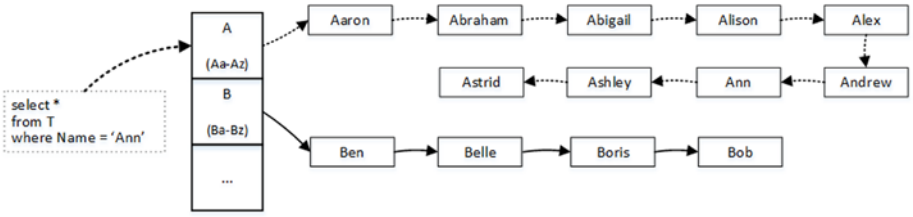


Figure 4-3. *Hash table lookup: 26 buckets*

The same principle applies to the hash indexes where choosing an incorrect number of buckets can lead to serious performance issues.

Let's create two non-durable memory-optimized tables, and populate them with 1,000,000 rows each, as shown in Listing 4-1. Both tables have exactly the same schema with a primary key constraint defined as the hash index. The number of buckets in the index is controlled by the bucket_count property. Internally, however, SQL Server rounds the provided value to the next power of two, so the HashIndex_HighBucketCount table would have 1,048,576 buckets in the index and the HashIndex_LowBucketCount table would have 1,024 buckets.

Listing 4-1. Bucket_count and Performance: Creating Memory-Optimized Tables

```
create table dbo.HashIndex_LowBucketCount
(
    Id int not null
        constraint PK_HashIndex_LowBucketCount
        primary key nonclustered
        hash with (bucket_count=1000),
    Value int not null
)
with (memory_optimized=on, durability=schema_only);

create table dbo.HashIndex_HighBucketCount
(
    Id int not null
        constraint PK_HashIndex_HighBucketCount
        primary key nonclustered
```

```
        hash with (bucket_count=1000000),
    Value int not null
)
with (memory_optimized=on, durability=schema_only);
go

;with N1(C) as (select 0 union all select 0) -- 2 rows
,N2(C) as (select 0 from N1 as t1 cross join N1 as t2) -- 4 rows
,N3(C) as (select 0 from N2 as t1 cross join N2 as t2) -- 16 rows
,N4(C) as (select 0 from N3 as t1 cross join N3 as t2) -- 256 rows
,N5(C) as (select 0 from N4 as t1 cross join N4 as t2) -- 65,536 rows
,N6(C) as (select 0 from N5 as t1 cross join N3 as t2) -- 1,048,576 rows
,Ids(Id) as (select row_number() over (order by (select null)) from N6)
insert into dbo.HashIndex_HighBucketCount(Id, Value)
    select Id, Id
    from ids
    where Id <= 1000000;

;with N1(C) as (select 0 union all select 0) -- 2 rows
,N2(C) as (select 0 from N1 as t1 cross join N1 as t2) -- 4 rows
,N3(C) as (select 0 from N2 as t1 cross join N2 as t2) -- 16 rows
,N4(C) as (select 0 from N3 as t1 cross join N3 as t2) -- 256 rows
,N5(C) as (select 0 from N4 as t1 cross join N4 as t2) -- 65,536 rows
,N6(C) as (select 0 from N5 as t1 cross join N3 as t2) -- 1,048,576 rows
,Ids(Id) as (select row_number() over (order by (select null)) from N6)
insert into dbo.HashIndex_LowBucketCount(Id, Value)
    select Id, Id
    from ids
    where Id <= 1000000;
```

Table 4-1 shows the execution time of the INSERT statements in my test environment. As you can see, inserting data into the HashIndex_HighBucketCount table is about 27 times faster compared to the HashIndex_LowBucketCount counterpart.

Table 4-1. *Execution Time of INSERT Statements*

dbo.HashIndex_HighBucketCount (1,048,576 buckets)	dbo.HashIndex_LowBucketCount (1024 buckets)
3,578 ms	99,312 ms

Listing 4-2 shows the query that returns the bucket count and row chains information using the sys.dm_db_xtp_hash_index_stats view. Keep in mind that this view scans the entire table, which is time consuming when the tables are large.

Listing 4-2. Obtaining Information About Hash Indexes

```
select
    s.name + '.' + t.name as [Table]
    ,i.name as [Index]
    ,stat.total_bucket_count as [Total Buckets]
    ,stat.empty_bucket_count as [Empty Buckets]
    ,floor(100. * empty_bucket_count / total_bucket_count)
        as [Empty Bucket %]
    ,stat.avg_chain_length as [Avg Chain]
    ,stat.max_chain_length as [Max Chain]
from
    sys.dm_db_xtp_hash_index_stats stat
        join sys.tables t on
            stat.object_id = t.object_id
        join sys.indexes i on
            stat.object_id = i.object_id and
            stat.index_id = i.index_id
        join sys.schemas s on
            t.schema_id = s.schema_id
```

Figure 4-4 shows the output of the query. As you can see, the `HashIndex_HighBucketCount` table has on average one row in the row chains, while the `HashIndex_LowBucketCount` table has almost a thousand rows per chain. It is worth noting that even though the hash function used by In-Memory OLTP provides relatively good random data distribution, some level of hash collision is still present.

	Table	Index	total_bucket_count	empty_bucket_count	avg_chain_length	max_chain_length
1	dbo.HashIndex_HighBucketCount	PK_HashIndex_HighBucketCount	1048576	398369	1	8
2	dbo.HashIndex_LowBucketCount	PK_HashIndex_LowBucketCount	1024	0	976	1035

Figure 4-4. *sys.dm_db_xtp_hash_index_stats output*

The incorrect bucket count estimation and long row chains can significantly affect performance of both reader and writer queries. You have already seen the performance impact for the insert operation. Now let's look at a select query.

Listing 4-3 shows the code that triggers 65,536 *Index Seek* operations in each memory-optimized table. I wrote this query in very inefficient way just to be able to demonstrate the impact of the long row chains.

Listing 4-3. Bucket_count and Performance: Selecting Data in the Tables

```
declare
    @T table(Id int not null primary key)

;with N1(C) as (select 0 union all select 0) -- 2 rows
,N2(C) as (select 0 from N1 as t1 cross join N1 as t2) -- 4 rows
,N3(C) as (select 0 from N2 as t1 cross join N2 as t2) -- 16 rows
```

```
,N4(C) as (select 0 from N3 as t1 cross join N3 as t2) -- 256 rows
,N5(C) as (select 0 from N4 as t1 cross join N4 as t2) -- 65,536 rows
,Ids(Id) as (select row_number() over (order by (select null)) from N5)
insert into @T(Id)
    select Id from Ids;

select t.id, c.Cnt
from @T t
    cross apply
     (
            select count(*) as Cnt
            from dbo.HashIndex_HighBucketCount h
            where h.Id = t.Id
    ) c;

select t.id, c.Cnt
from @T t
    cross apply
     (
            select count(*) as Cnt
            from dbo.HashIndex_LowBucketCount h
            where h.Id = t.Id
    ) c;
```

You can confirm that the queries traversed the row chains 65,536 times by analyzing the execution plan shown in Figure 4-5.

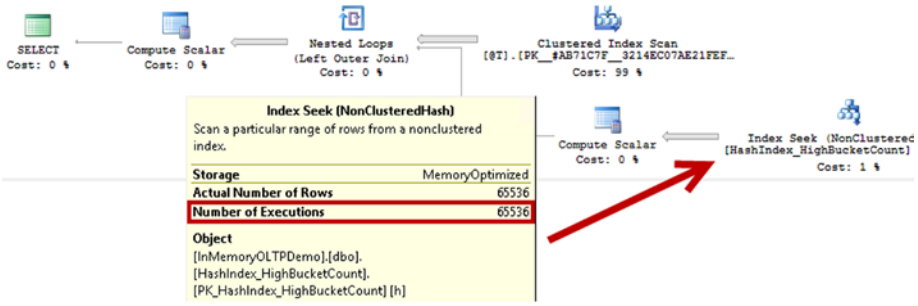


Figure 4-5. *Execution plan of the queries*

Table 4-2 shows the queries' execution time in my environment where the query against `HashIndex_LowBucketCount` table was about 20 times slower.

Table 4-2. *Execution Time of SELECT Statements*

dbo.HashIndex_HighBucketCount (1,048,576 buckets)	dbo.HashIndex_LowBucketCount (1024 buckets)
644 ms	13,524 ms

While you can clearly see that underestimation of the bucket counts can degrade system performance, overestimation is not good either. First, every bucket uses 8 bytes to store the memory pointer, and a large number of unused buckets is a waste of precious system memory. For example, defining the index with `bucket_count=100000000` will introduce 134,217,728 buckets, which will require 128MB of RAM. This does not seem much in the scope of the single index; however, it could become an issue as the number of indexes grows up.

Moreover, SQL Server needs to scan all buckets in the index when it performs an *Index Scan* operation, and extra buckets add some overhead to the process. Listing 4-4 shows the queries that demonstrate such an overhead.

Listing 4-4. Bucket_count and Performance: Index Scan Queries

```
select count(*) from dbo.HashIndex_HighBucketCount;
select count(*) from dbo.HashIndex_LowBucketCount;
```

Table 4-3 shows the execution time in my environment. As you see, the overhead of scanning extra buckets is not significant; however, it still exists.

Table 4-3. *Execution Time of SELECT Statements*

dbo.HashIndex_HighBucketCount (1,048,576 buckets)	dbo.HashIndex_LowBucketCount (1024 buckets)
313 ms	280 ms

Choosing the Right Bucket Count

Choosing the right number of buckets in a hash index is a tricky but very important subject. To make matters worse, you have to make the right decision at the design stage; it is impossible to alter the index and change the `bucket_count` once a table is created.

In ideal situation, you should have the number of buckets that would exceed cardinality (number of unique keys) of the index. Obviously, you should take future system growth and projected workload changes into consideration. It is not a good idea to create an index based on the current data cardinality if you expect the system to handle much more data in the future.

■ **Note** Microsoft suggests setting the `bucket_count` to be between one and two times the number of distinct values in the index. You can read more at https://msdn.microsoft.com/en-us/library/dn494956.aspx.

Low-cardinality columns with a large number of duplicated values are usually bad candidates for hash indexes. The same data values generate the same hash and, therefore, rows will be linked to long row chains. Obviously, there are always exceptions, and you should analyze the queries and workload in your system, taking into consideration the data modification overhead introduced by the long row chains.

In existing indexes, you can analyze the output of the `sys.dm_db_xpt_hash_index_stats` view to determine if the number of buckets in the index is sufficient. If the number of empty buckets is less than 10 percent of the total number of buckets in the index, the bucket count is likely to be too low. Ideally, at least 33 percent of the buckets in the index should be empty.

With all that being said, it is often better to err on the side of caution and overestimate rather than underestimate the number. Even though overestimation impacts the performance of the *Index Scan*, this impact is much lower compared to the one introduced by long row chains. Obviously, you need to remember that every bucket uses 8 bytes of memory whether it is empty or not.

Changing the Bucket Count in the Index

As you already know, it is impossible to alter the table and change the `bucket_count` in the index after the table is created. The only option of changing it is to recreate the table, which is impossible to do while keeping the table online.

To make matter worse, the `sp_rename` stored procedure does not work with memory-optimized tables. It is impossible to create a new memory-optimized table with the desired structure, dump data there, and drop an old table and rename a new table afterwards. You will need to recreate an old table and copy data the second time if you want to keep the table name intact.

■ **Tip** You can use synonyms referencing the new table under the old table name, making it transparent to the code. You can read more about synonyms at https://msdn.microsoft.com/en-us/library/ms187552.aspx.

When you want to keep the table name intact, you can export data to and import data from the flat files using the `bcp` utility. Alternatively, you can create and use either an on-disk or memory-optimized table as a temporary staging place.

Obviously, a memory-optimized table is the faster choice compared to an on-disk table; however, you should consider memory requirements during the process. Even though the garbage collector eventually deallocates deleted rows from the memory, it would not happen instantly after you dropped the table. You should have enough memory to accommodate at least two extra copies of the data to be on the safe side.

Do not create any unnecessary indexes on the staging table. Use heaps in the case of an on-disk table or a single primary key constraint with a memory-optimized table. This will help you to reduce memory consumption and speed up the process.

Finally, avoid using non-durable memory-optimized tables to stage the data. Even though this could significantly speed up the process and reduce transaction log overhead, you can lose the data if an unexpected crash or failover occurred during the data movement.

■ **Tip** You can reduce transaction log overhead by staging data in another database temporarily created for such a purpose. You will still write the data to transaction log in the staging database; however, those log records won't need to be backed up or transmitted over the network to the secondary servers. It is also beneficial to use the `SELECT INTO` operator when copying data into an on-disk table to make the operation minimally logged.

Hash Indexes and SARGability

In the database world, queries are treated as SARGable (Search ARGument Able) when they and their predicates allow the Database Engine to utilize *Index Seek* operations during query execution.

Hash indexes have different SARGability rules as compared to B-Tree indexes defined on on-disk tables. They are efficient only in the case of a **point-lookup equality search**, which allows SQL Server to calculate the corresponding hash value of the index key(s) and find a bucket that references the desired chain of rows.

In the case of composite hash indexes, SQL Server calculates the hash value for the combined value of all key columns. A hash value calculated on a subset of the key columns would be different and, therefore, a query should have equality predicates on all key columns for the index to be useful.

This behavior is different from indexes on on-disk tables. Consider the situation where you want to define an index on `(LastName, FirstName)` columns. In the case of on-disk tables, that index can be used for an *Index Seek* operation, regardless of whether the predicate on the `FirstName` column is specified in the `where` clause of a query. Alternatively, a composite hash index on a memory-optimized table requires queries to have equality predicates on both `LastName` and `FirstName` in order to calculate a hash value that allows for choosing the right hash bucket in the index.

Let's create on-disk and memory-optimized tables with composite indexes on the `(LastName, FirstName)` columns, populating them with the same data as shown in Listing 4-5.

Listing 4-5. Composite Hash Index: Test Tables Creation

```
create table dbo.CustomersOnDisk
(
    CustomerId int not null identity(1,1),
    FirstName varchar(64) collate Latin1_General_100_BIN2 not null,
    LastName varchar(64) collate Latin1_General_100_BIN2 not null,
    Placeholder char(100) null,

    constraint PK_CustomersOnDisk
    primary key clustered(CustomerId)
);

create nonclustered index IDX_CustomersOnDisk_LastName_FirstName
on dbo.CustomersOnDisk(LastName, FirstName)
go

create table dbo.CustomersMemoryOptimized
(
    CustomerId int not null identity(1,1)
        constraint PK_CustomersMemoryOptimized
        primary key nonclustered
        hash with (bucket_count = 30000),
    FirstName varchar(64) collate Latin1_General_100_BIN2 not null,
    LastName varchar(64) collate Latin1_General_100_BIN2 not null,
    Placeholder char(100) null,

    index IDX_CustomersMemoryOptimized_LastName_FirstName
    nonclustered hash(LastName, FirstName)
    with (bucket_count = 1024),
)
with (memory_optimized = on, durability = schema_only)
go

-- Inserting cross-joined data for all first and last names 50 times
-- using GO 50 command in Management Studio
;with FirstNames(FirstName)
as
(
    select Names.Name
    from
    (
        values('Andrew'),('Andy'),('Anton'),('Ashley'),('Boris'),
        ('Brian'),('Cristopher'),('Cathy'),('Daniel'),('Donny'),
        ('Edward'),('Eddy'),('Emy'),('Frank'),('George'),('Harry'),
        ('Henry'),('Ida'),('John'),('Jimmy'),('Jenny'),('Jack'),
        ('Kathy'),('Kim'),('Larry'),('Mary'),('Max'),('Nancy'),
```

```
        ('Olivia'),('Paul'),('Peter'),('Patrick'),('Robert'),
        ('Ron'),('Steve'),('Shawn'),('Tom'),('Timothy'),
        ('Uri'),('Vincent')
    ) Names(Name)
)
,LastNames(LastName)
as
(
    select Names.Name
    from
    (
        values('Smith'),('Johnson'),('Williams'),('Jones'),('Brown'),
            ('Davis'),('Miller'),('Wilson'),('Moore'),('Taylor'),
            ('Anderson'),('Jackson'),('White'),('Harris')
    ) Names(Name)
)
insert into dbo.CustomersOnDisk(LastName, FirstName)
    select LastName, FirstName
    from FirstNames cross join LastNames
go 50

insert into dbo.CustomersMemoryOptimized(LastName, FirstName)
    select LastName, FirstName
    from dbo.CustomersOnDisk;
```

For the first test, let's run select statements against both tables, specifying both `LastName` and `FirstName` as predicates in the queries, as shown in Listing 4-6.

Listing 4-6. Composite Hash Index: Selecting Data Using Both Index Columns as Predicates

```
select CustomerId, FirstName, LastName
from dbo.CustomersOnDisk
where FirstName = 'Paul' and LastName = 'White';

select CustomerId, FirstName, LastName
from dbo.CustomersMemoryOptimized
where FirstName = 'Paul' and LastName = 'White';
```

As you can see in Figure 4-6, SQL Server is able to use an *Index Seek* operation in both cases.

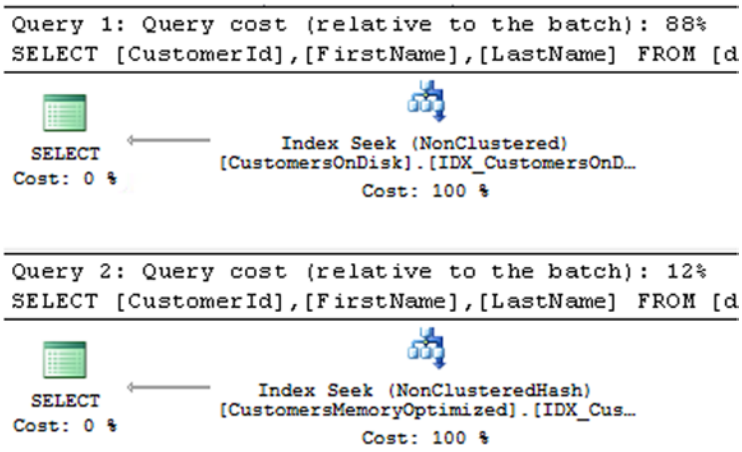


Figure 4-6. *Composite hash index: execution plans when queries use both index columns as predicates*

In the next step, let's check what happens if you remove the filter by `FirstName` from the queries. The code is shown in Listing 4-7.

Listing 4-7. Composite Hash Index: Selecting Data Using the Leftmost Index Column Only

```
select CustomerId, FirstName, LastName
from dbo.CustomersOnDisk
where LastName = 'White';

select CustomerId, FirstName, LastName
from dbo.CustomersMemoryOptimized
where LastName = 'White';
```

In the case of the on-disk index, SQL Server is still able to utilize an *Index Seek* operation. This is not the case for the composite hash index defined on the memory-optimized table. You can see the execution plans for the queries in Figure 4-7.

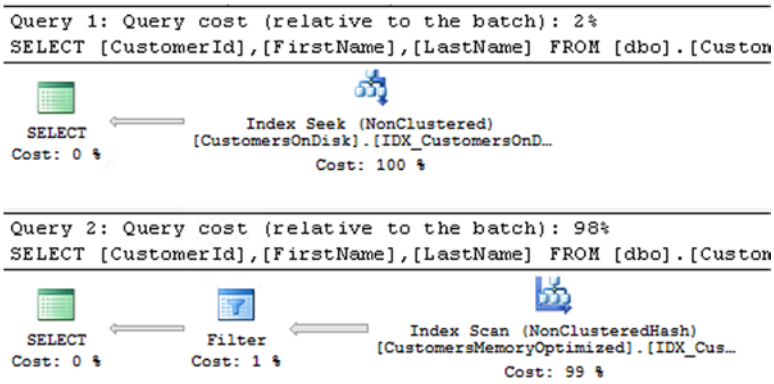


Figure 4-7. *Composite hash index: execution plans when queries use the leftmost index column only*

Statistics on Memory-Optimized Tables

Even though SQL Server creates index- and column-level statistics on memory-optimized tables, it does not update the statistics automatically. This behavior leads to a very interesting situation: indexes on memory-optimized tables are created with the tables and, therefore, the statistics are created at the time when the tables are empty and are never updated automatically afterwards.

You can validate it by running the `DBCC SHOW_STATISTICS` statement shown in Listing 4-8.

Listing 4-8. Analyzing Index Statistics

```
dbcc show_statistics
(
    'dbo.HashIndex_HighBucketCount'
    ,'PK_HashIndex_HighBucketCount'
)
```

The output shown in Figure 4-8 illustrates that the statistics is empty.

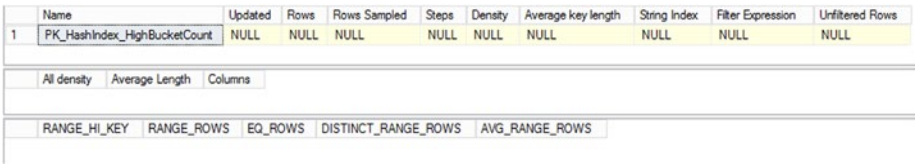


Figure 4-8. *Output of DBCC SHOW_STATISTICS statement*

You need to keep this behavior in mind while designing a statistics maintenance strategy in the system. You should update the statistics after the data is loaded into the table when SQL Server or the database restarts. Moreover, if the data in a memory-optimized table is volatile, which is usually the case, you should manually update statistics on a regular basis.

You can update individual statistics with the `UPDATE STATISTICS` command. Alternatively, you can use the `sp_updatestats` stored procedure to update all statistics in the database. The `sp_updatestats` stored procedure always updates all statistics on memory-optimized tables, which is different from how it works for on-disk tables, where such a stored procedure skips statistics that do not need to be updated.

SQL Server always performs a full scan while updating statistics on memory-optimized tables. This behavior is also different from on-disk tables, whereas SQL Server samples the data by default. Finally, you need to specify the `NORECOMPUTE` option when you run `CREATE STATISTICS` or `UPDATE STATISTICS` statements. Listing 4-9 shows an example.

Listing 4-9. Updating Statistics on Memory-Optimized Table

```
update statistics dbo.HashIndex_HighBucketCount
with fullscan, norecompute;
```

If you run the `DBCC SHOW_STATISTICS` statement from Listing 4-8 again, you should see that the statistics have been updated (see Figure 4-9).

	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows
1	PK_HashIndex_HighBucketCount	Feb 14 2015 1:06PM	1000000	1000000	3	1	4	NO	NULL	1000000

	All density	Average Length	Columns
1	1E-06	4	Id

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	1	0	1	0	1
2	999999	999997	1	999997	1
3	1000000	0	1	0	1

Figure 4-9. *Output of DBCC SHOW_STATISTICS statement after the statistics are updated*

Missing statistics can introduce suboptimal execution plans with the nested loop joins when SQL Server chooses inner and outer inputs for the operator. As you know, the nested loop join algorithm processes the inner input for every row from the outer input, and it is more efficient to put smaller input to the outer side. Listing 4-10 shows the algorithm for the inner nested loop join as the reference.

Listing 4-10. Inner Nested Loop Join Algorithm

```
for each row R1 in outer table
    for each row R2 in inner table
        if R1 joins with R2
            return join (R1, R2)
```

Missing statistics can lead to a situation when SQL Server choses the inner and outer inputs incorrectly, which can lead to highly inefficient plans.

Let's create two tables, populating them with some data, as shown in Listing 4-11.

Listing 4-11. Missing Statistics and Inefficient Execution Plans: Table Creation

```
create table dbo.T1
(
    ID int not null identity(1,1)
        primary key nonclustered hash
        with (bucket_count = 8192),
    T1Col int not null,
    Placeholder char(100) not null
        constraint DEF_T1_Placeholder
        default('1'),

    index IDX_T1Col
    nonclustered hash(T1Col)
    with (bucket_count = 1024)
)
with (memory_optimized = on, durability = schema_only);

create table dbo.T2
(
    ID int not null identity(1,1)
        primary key nonclustered hash
        with (bucket_count = 8192),
    T2Col int not null,
    Placeholder char(100) not null
        constraint DEF_T2_Placeholder
        default('2'),

    index IDX_T2Col
    nonclustered hash(T2Col)
    with (bucket_count = 1024)
)
with (memory_optimized = on, durability = schema_only);

;with N1(C) as (select 0 union all select 0) -- 2 rows
,N2(C) as (select 0 from N1 as t1 cross join N1 as t2) -- 4 rows
,N3(C) as (select 0 from N2 as t1 cross join N2 as t2) -- 16 rows
,N4(C) as (select 0 from N3 as t1 cross join N3 as t2) -- 256 rows
,N5(C) as (select 0 from N4 as t1 cross join N3 as t2) -- 4,096 rows
,Ids(Id) as (select row_number() over (order by (select null)) from N5)
    insert into dbo.T1(T1Col)
        select 1 from Ids;

insert into dbo.T2(T2Col)
    select -1 from dbo.T1;

update dbo.T1
set T1Col = 2
where ID = 4096;

update dbo.T2
set T2Col = -2
where ID = 1;
```

The data in both tables distributed unevenly. You can confirm it by running the query in Listing 4-12. Figure 4-10 illustrates the data distribution in the tables.

Listing 4-12. Missing Statistics and Inefficient Execution Plans: Checking Data Distribution in the Tables

```
select 'T1' as [Table], T1Col as [Value], count(*) as [Count]
from dbo.T1
group by T1Col

union all

select 'T2' as [Table], T2Col as [Value], count(*) as [Count]
from dbo.T2
group by T2Col;
```

	Table	Value	Count
1	T1	1	4095
2	T1	2	1
3	T2	-2	1
4	T2	-1	4095

Figure 4-10. *Missing statistics and inefficient execution plans: data distribution*

As the next step, let's run two queries that join the data from the tables as it is shown in Listing 4-13. Both queries will return just a single row.

Listing 4-13. Missing Statistics and Inefficient Execution Plans: Test Queries

```
select *
from dbo.T1 t1 join dbo.T2 t2 on
    t1.ID = t2.ID
where
    t1.T1Col = 2 and
    t2.T2Col = -1;

select *
from dbo.T1 t1 join dbo.T2 t2 on
    t1.ID = t2.ID
where
    t1.T1Col = 1 and
    t2.T2Col = -2
```

As you can see in Figure 4-11, SQL Server generated identical execution plans for both queries using the T1 table in the outer part of the join. This plan is very efficient for the first query; there is the only one row with `T1Col = 2` and, therefore, SQL Server had to perform an inner input lookup just once. Unfortunately, it is not the case for the second query, which leads to 4,095 *Index Seek* operations on the T2 table.

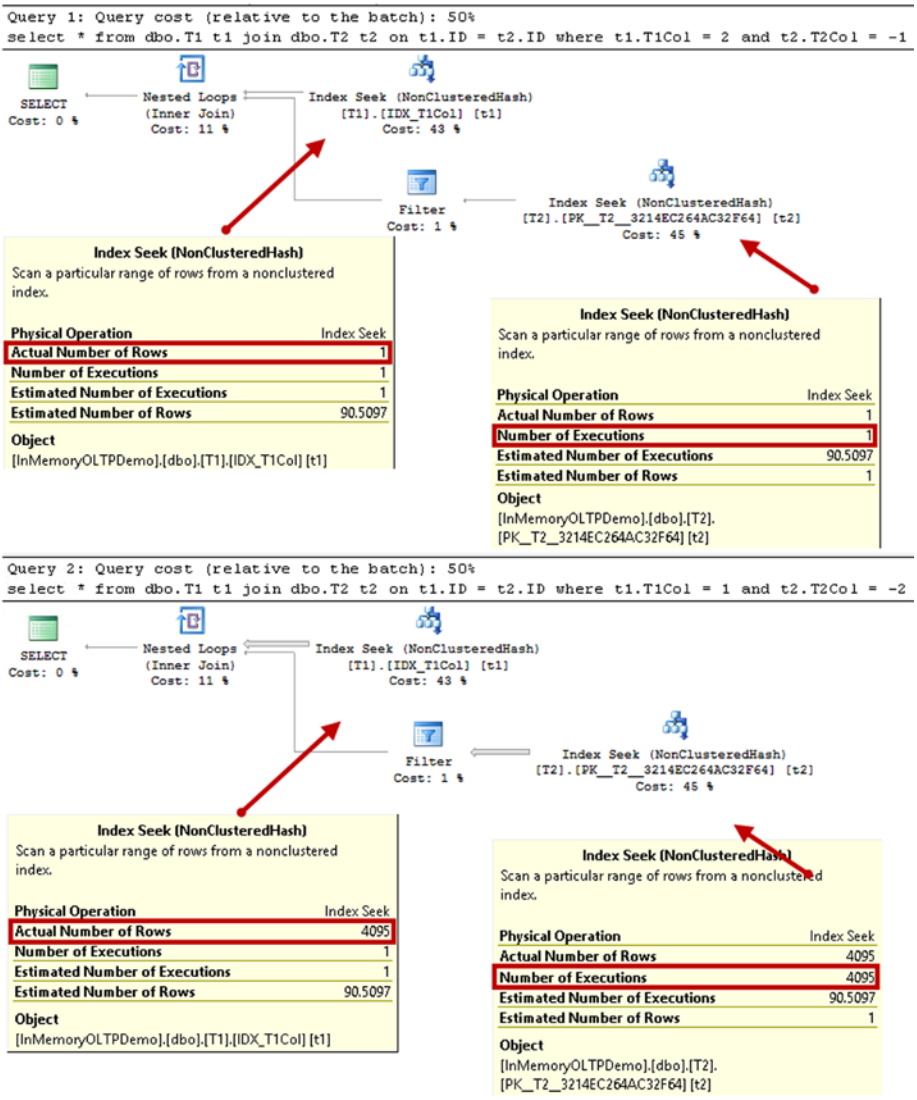


Figure 4-11. *Missing statistics and inefficient execution plans: execution plans*

Let's update statistics on both tables, as shown in Listing 4-14.

Listing 4-14. Missing Statistics and Inefficient Execution Plans: Updating Statistics

```
update statistics dbo.T1 with fullscan, norecompute;
update statistics dbo.T2 with fullscan, norecompute;

dbcc show_statistics('dbo.T1','IDX_T1Col');
dbcc show_statistics('dbo.T2','IDX_T2Col');
```

Figure 4-12 illustrates that the statistics have been updated.

	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows
1	IDX_T1Col	May 16 2015 10:22AM	4096	4096	2	0	4	NO	NULL	4096

	All density	Average Length	Columns
1	0.5	4	T1Col

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	1	0	4095	0	1
2	2	0	1	0	1

	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows
1	IDX_T2Col	May 16 2015 10:22AM	4096	4096	2	0	4	NO	NULL	4096

	All density	Average Length	Columns
1	0.5	4	T2Col

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	-2	0	1	0	1
2	-1	0	4095	0	1

Figure 4-12. *Missing statistics and inefficient execution plans: index statistics*

Now, if you run the queries from Listing 4-13 again, SQL Server can generate an efficient execution plan for the second query, as shown in Figure 4-13.

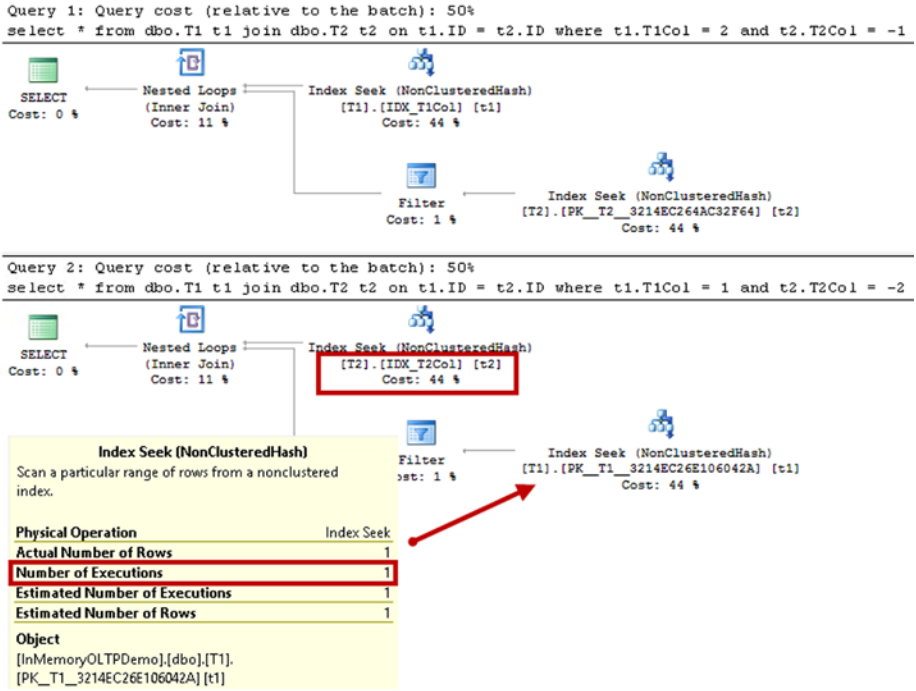


Figure 4-13. *Missing statistics and inefficient execution plans: execution plans after statistics update*

■ **Note** You can read more about statistics on memory-optimized tables at http://msdn.microsoft.com/en-us/library/dn232522.aspx.

Summary

Hash indexes consist of an array of hash buckets, each of which stores the pointer to the chain of rows with the same index key column(s) hash. Hash indexes help to optimize point-lookup operations when queries search for the rows using equality predicates. In case of composite hash indexes, the query should have equality predicates on all key columns for the index to be useful.

Choosing the right bucket count is extremely important. Underestimations lead to long row chains, which could seriously degrade performance of the queries. Overestimations increase memory consumption and decrease performance of the index scans.

Low-cardinality columns lead to the long row chains and are usually bad candidates for hash indexes.

You should analyze index cardinality and consider future system growth when choosing the right bucket count. Ideally, you should have at least 33 percent of buckets empty. You can get information about buckets and row chains with the `sys.dm_db_xtp_hash_index_stats` view.

SQL Server creates statistics on the indexes on memory-optimized tables; however, statistics are not updated automatically. You should update statistics manually using the `UPDATE STATISTICS` statement or the `sp_updatestats` procedure on a regular basis.

CHAPTER 5

Nonclustered Indexes

This chapter discusses nonclustered indexes, which is the second type of indexes supported by the In-Memory OLTP Engine. It shows how to define nonclustered indexes, talks about their SARGability rules, and explains their internal structure.

Working with Nonclustered Indexes

Nonclustered indexes are another type of indexes supported by the In-Memory OLTP Engine. In contrast to hash indexes, which are optimized to support point-lookup equality searches, nonclustered indexes help you search data based on a range of values. They have a somewhat similar structure to regular indexes on on-disk tables. They are not exactly the same, however, and I will discuss their internal implementation in depth later in this chapter.

TERMINOLOGY ISSUE

Nonclustered indexes were introduced in SQL Server 2014 CTP 2, and the documentation and whitepapers for that version used the term "range indexes" to reference them. However, in the production release of SQL Server 2014, Microsoft changed the terminology to "nonclustered indexes."

That terminology can be confusing because hash indexes are also not clustered. In fact, the concepts of *heaps* and *clustered indexes* cannot be applied to In-Memory OLTP. Data rows are not stored in any particular order nor are they grouped together on the data pages in memory.

It is also worth mentioning that the minimal `index_id` value of In-Memory OLTP indexes is 2, which corresponds to nonclustered indexes in on-disk tables.

Creating Nonclustered Indexes

Nonclustered indexes are created inline as part of the `CREATE TABLE` statement. The syntax is similar to hash index creation; however, you should omit the keyword `HASH` and you do not need to specify the number of buckets in the index properties. Collation requirement still exists; you cannot index text data unless a column uses `BIN2` binary collation.

The code in Listing 5-1 creates a memory-optimized table with two nonclustered indexes, one composite and another on the single column.

Listing 5-1. Creating a Table with Two Nonclustered Indexes

```
create table dbo.Customers
(
    CustomerId int identity(1,1) not null
        constraint PK_Customers
        primary key nonclustered
        hash with (bucket_count=1000),
    FirstName varchar(32)
        collate Latin1_General_100_BIN2 not null,
    LastName varchar(64)
        collate Latin1_General_100_BIN2 not null,
    FullName varchar(97)
        collate Latin1_General_100_BIN2 not null,

    index IDX_LastName_FirstName
    nonclustered(LastName, FirstName),

    index IDX_FullName
    nonclustered(FullName)
)
with (memory_optimized=on, durability=schema_only);
```

Using Nonclustered Indexes

Similar to B-Tree indexes in on-disk tables, the data in nonclustered indexes is sorted accordingly to the value of index key columns. As result, nonclustered indexes are beneficial in a large number of use cases. They can lead to an *Index Seek* operation in scenarios when query predicates allow SQL Server to locate and isolate a subset of the index keys for processing. With very few exceptions, the SARGability rules for nonclustered indexes match the rules for indexes defined on on-disk tables.

Listing 5-2 shows several queries against the `dbo.Customers` table. SQL Server is able to use *Index Seek* with all of them.

Listing 5-2. Queries That Lead to Index Seek Operations

```
-- Point-Lookup specifying all columns in the index
select CustomerId, FirstName, LastName
from dbo.Customers
where LastName = 'White' and FirstName = 'Paul';

-- Point-lookup using leftmost index column
select CustomerId, FirstName, LastName
from dbo.Customers
where LastName = 'White';

-- Using ">", ">=", "<", "<=" comparison
select CustomerId, FirstName, LastName
from dbo.Customers
where LastName > 'White';

-- Prefix Search
select CustomerId, FirstName, LastName
from dbo.Customers
where LastName like 'Wh%';

-- IN list
select CustomerId, FirstName, LastName
from dbo.Customers
where LastName in ('White','Moore');
```

Similar to B-Tree indexes, *Index Seek* is impossible when query predicates do not allow isolating a subset of the index keys for processing. Listing 5-3 shows several examples of such queries.

Listing 5-3. Queries That Lead to Index Scan Operations

```
-- Omitting left-most index column(s)
select CustomerId, FirstName, LastName
from dbo.Customers
where FirstName = 'Paul';

-- Substring Search
select CustomerId, FirstName, LastName
from dbo.Customers
where LastName like '%hit%';

-- Functions
select CustomerId, FirstName, LastName
from dbo.Customers
where len(LastName) = 5;
```

As the opposite of B-Tree indexes on on-disk tables, nonclustered indexes are unidirectional, and SQL Server is unable to scan index keys in the order opposite of how they were sorted. You should keep this behavior in mind when you define an index and choose the sorting order for the columns.

Let's illustrate that with an example; we'll create an on-disk table with the same structure as `dbo.Customers`, and populate both tables with the same data. Listing 5-4 shows the code to do so.

Listing 5-4. Nonclustered Indexes and Sorting Order: On-disk Table Creation

```
create table dbo.Customers_OnDisk
(
    CustomerId int identity(1,1) not null,
    FirstName varchar(32) not null,
    LastName varchar(64) not null,
    FullName varchar(97) not null,

    constraint PK_Customers_OnDisk
    primary key clustered(CustomerId)
);

create nonclustered index IDX_Customers_OnDisk_LastName_FirstName
on dbo.Customers_OnDisk(LastName, FirstName);

create nonclustered index IDX_Customers_OnDisk_FullName
on dbo.Customers_OnDisk(FullName);
go

;with FirstNames(FirstName)
as
(
    select Names.Name
    from
    (
        values('Andrew'),('Andy'),('Anton'),('Ashley'),('Boris'),
        ('Brian'),('Cristopher'),('Cathy'),('Daniel'),('Don'),
        ('Edward'),('Eddy'),('Emy'),('Frank'),('George'),('Harry'),
        ('Henry'),('Ida'),('John'),('Jimmy'),('Jenny'),('Jack'),
        ('Kathy'),('Kim'),('Larry'),('Mary'),('Max'),('Nancy'),
        ('Olivia'),('Paul'),('Peter'),('Patrick'),('Robert'),
        ('Ron'),('Steve'),('Shawn'),('Tom'),('Timothy'),
        ('Uri'),('Vincent')
    ) Names(Name)
)
,LastNames(LastName)
as
(
    select Names.Name
```

```
    from
    (
        values('Smith'),('Johnson'),('Williams'),('Jones'),('Brown'),
        ('Davis'),('Miller'),('Wilson'),('Moore'),('Taylor'),
        ('Anderson'),('Jackson'),('White'),('Harris')
    ) Names(Name)
)
insert into dbo.Customers(LastName, FirstName, FullName)
    select LastName, FirstName, FirstName + ' ' + LastName
    from FirstNames cross join LastNames;

insert into dbo.Customers_OnDisk(LastName, FirstName, FullName)
    select LastName, FirstName, FullName
    from dbo.Customers;
```

Let's run the queries that select several rows in ascending order, which match the index sorting order. The queries are shown in Listing 5-5.

Listing 5-5. Nonclustered Indexes and Sorting Order: Selecting Data in the Same Order with the Index Key Column

```
select top 3 CustomerId, FirstName, LastName, FullName
from dbo.Customers_OnDisk
order by FullName ASC;

select top 3 CustomerId, FirstName, LastName, FullName
from dbo.Customers
order by FullName ASC;
```

Figure 5-1 shows the execution plans for the queries. SQL Server scans the indexes starting with the lowest key and stops after it read three rows. The execution plans are similar for both queries with the exception of required *Key Lookup* with on-disk data. SQL Server uses it to obtain the values of the `FirstName` and `LastName` columns from the clustered index of the table.

Key Lookup is not required with memory-optimized tables where the index pointers are part of the actual data rows and the indexes are covering the queries.

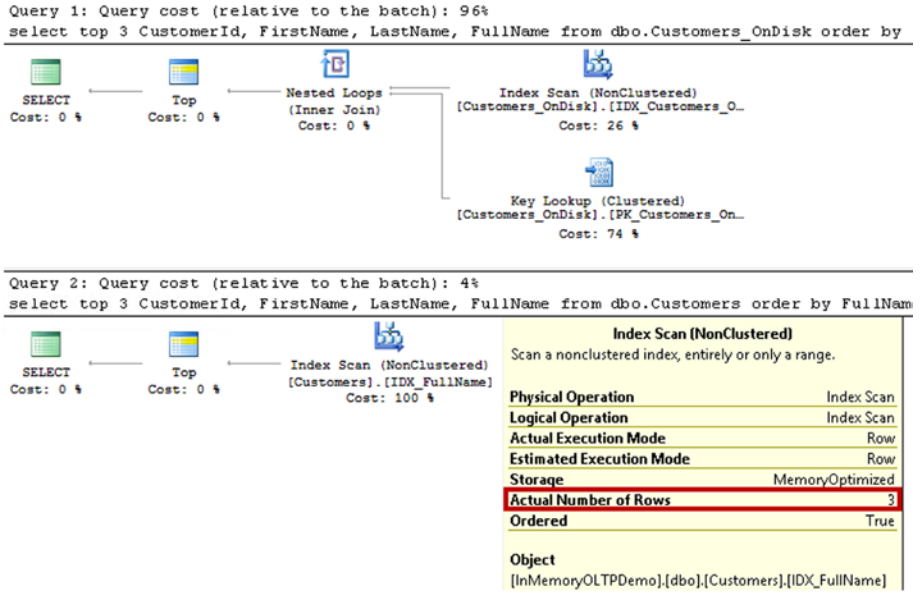


Figure 5-1. *Execution plans when the order by results match the index sorting order*

The situation changes if you need to sort the output in the descending order, as shown in Listing 5-6.

Listing 5-6. Nonclustered Indexes and Sorting Order: Selecting Data in the Opposite Order with Index Key Column

```
select top 3 CustomerId, FirstName, LastName, FullName
from dbo.Customers_OnDisk
order by FullName DESC;

select top 3 CustomerId, FirstName, LastName, FullName
from dbo.Customers
order by FullName DESC;
```

As you can see in Figure 5-2, SQL Server is able to scan the on-disk table index in the order opposite of how it was defined. However, this is not the case for memory-optimized tables where indexes are unidirectional. SQL Server decides to scan the primary key and sort the data afterwards.

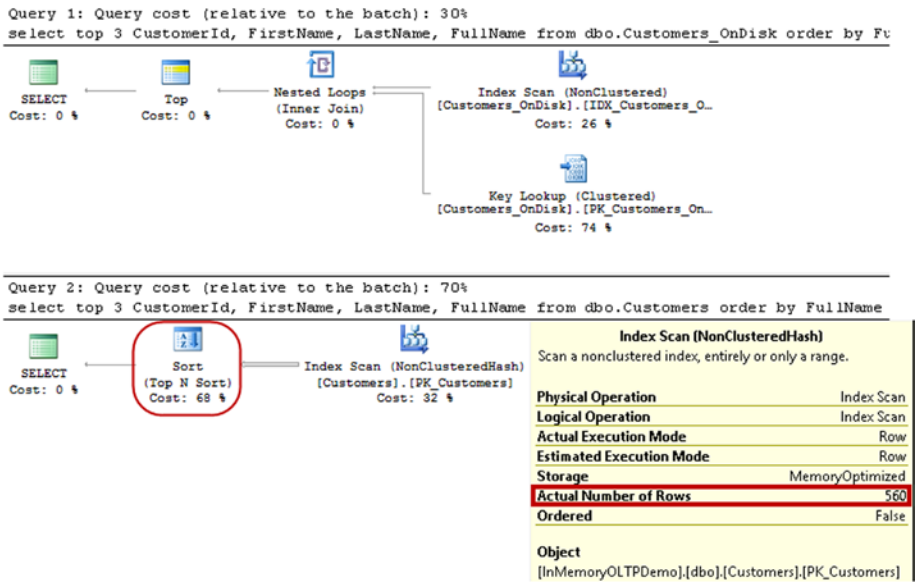


Figure 5-2. *Execution plans when the order by results are the opposite of the index sorting order*

Finally, index statistics limitations, which were discussed in Chapter 4, still apply to the nonclustered indexes. SQL Server creates statistics at the time of index creation; however, they are never updated automatically. You should update statistics manually on a regular basis.

Nonclustered Indexes Internals

Nonclustered indexes use a lock- and latch-free variation of B-Tree, called Bw-Tree, which was designed by Microsoft Research in 2011. Let's look at the Bw-Tree structure in detail.

Bw-Tree Overview

Similar to B-Trees, index pages in a Bw-Tree contain a set of ordered index key values. However, Bw-Tree pages do not have a fixed size and they are unchangeable after they are built. The maximum page size, however, is 8KB.

Rows from a leaf level of the nonclustered index contain the pointers to the data row chains with the same index key values. This works in a similar manner to hash indexes, when multiple rows and/or versions of a row are linked together. Each index in the table adds a pointer to the index pointer array in the row, regardless of its type: hash or nonclustered.

Root and intermediate levels in nonclustered indexes are called *internal pages*. Similar to B-Tree indexes, internal pages point to the next level in the index. However, instead of pointing to the actual data page, internal pages use a *logical page id* (PID),

which is a position (offset) in a separate array-like structure called a *mapping table*. In turn, each element in the mapping table contains a pointer to the actual index page.

Figure 5-3 shows an example of a nonclustered index and a mapping table. Each index row from the internal page stores the *highest* key value on the next-level page and PID. This is different from a B-Tree index, where intermediate- and root-level index rows store the *lowest* key value of the next-level page instead. Another difference is that the pages in a Bw-Tree are not linked in a double-linked list. Each page knows the PID of the next page on the same level and does not know PID of the previous page. Even though it appears as a pointer (arrow) in Figure 5-3, that link is done through the mapping table, similar to links to pages on the next level.

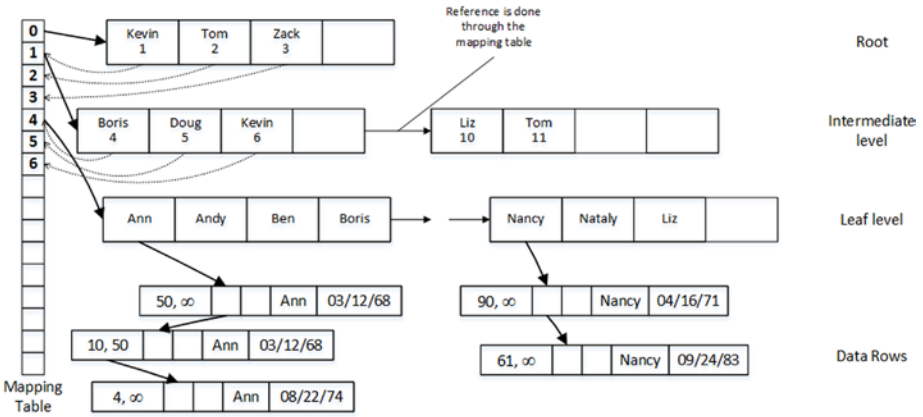


Figure 5-3. *Nonclustered index structure*

Even though a Bw-Tree looks very similar to a B-Tree, there is one conceptual difference: the leaf level of an on-disk B-Tree index consists of separate index rows for each data row in the index. If multiple data rows have the same key value, the index would have multiple leaf level rows with the same index key stored.

Alternatively, in-memory nonclustered indexes store one index row (pointer) to the row chain that includes all of the data rows that have the same key value. Only one index row (pointer) per key value is stored in the index. You can see this in Figure 5-3, where the leaf level of the index has single rows for the key values of `Ann` and `Nancy`, even though the row chain includes more than one data row for each value.

■ **Tip** You can compare the structure of B-Tree and Bw-Tree indexes by looking at Figures 3-1 and 3-2 from Chapter 3, which show clustered and nonclustered B-Tree indexes on on-disk tables.

Index Pages and Delta Records

As mentioned, pages in nonclustered indexes are unchangeable once they are built. SQL Server builds a new version of the page when it needs to be updated and replaces the page pointer in the mapping table, which avoids changing internal pages that reference an old (obsolete) page.

Every time SQL Server needs to change a leaf-level index page it creates one or two *delta* records that represent the changes. `INSERT` and `DELETE` operations generate a single insert or delete delta record, while an `UPDATE` operation generates two delta records, deleting old and inserting new values. Delta records create a chain of memory pointers with the last pointer to the actual index page. SQL Server also replaces a pointer in the mapping table with the address of the first delta record in the chain.

Figure 5-4 shows an example of a leaf-level page and delta records if the following actions occurred in the sequence: R1 index row is updated, R2 row is deleted, and R3 row is inserted.

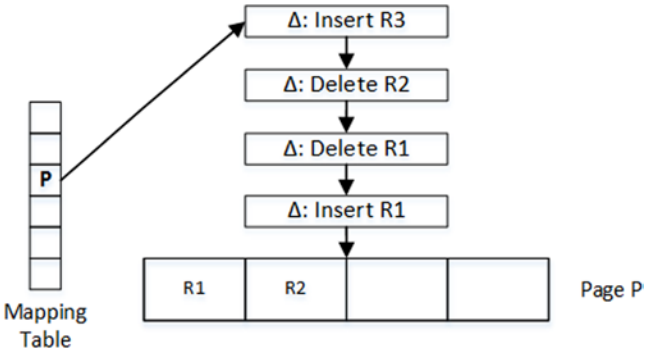


Figure 5-4. *Delta records and nonclustered index leaf page*

■ **Note** The internal implementation of the In-Memory OLTP Engine guarantees that multiple sessions cannot simultaneously update memory pointers in the various In-Memory OLTP objects, thereby overwriting each other's changes. This process is covered in detail in Appendix A.

The internal and leaf pages of nonclustered indexes consist of two areas: a *header* and *data*. The header area includes information about the page such as the following:

- **PID**: The position (offset) in the mapping table
- **Page type**: The type of the page, such as leaf, internal, delta, or special
- **Right page PID**: The position (offset) of the next page in the mapping table

- **Height**: The number of levels from the current page to the leaf level of the index
- **The number of key values** (index rows) stored on the page
- **Delta records statistics**: Includes the number of delta records and space used by the delta key values
- **The max value of a key** on the page

The data area of the page includes either two or three arrays depending on the index keys data types. The arrays are

- **Values**: An array of 8-byte pointers. Internal pages in the index store the PID of next-level pages. Leaf-level pages store pointers to the first row in the chain of rows with the corresponding key value. It is worth noting that even though PID requires 4 bytes to store a value, SQL Server uses 8-byte elements to preserve the same page structure between internal and leaf pages.
- **Keys**: An array of key values stored on the page.
- **Offsets**: An array of two-byte offsets where individual key values in keys array start. Offsets are stored only if keys have variable-length data.

Delta records, in a nutshell, are one-record index data pages. The structure of delta data pages is similar to the structure of internal and leaf pages. However, instead of arrays of values and keys, delta data pages store operation code (insert or delete) and a single key value and pointer to the first data row in a row chain.

Figure 5-5 shows an example of a leaf-level index page with an insert delta record.

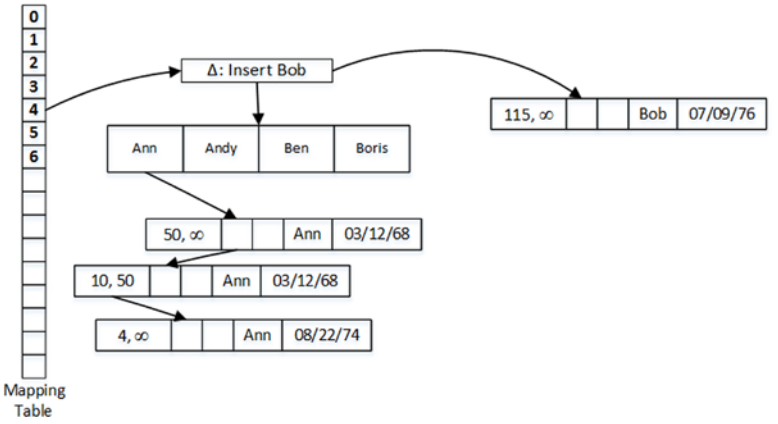


Figure 5-5. *A leaf-level index page with an insert delta record*

SQL Server needs to traverse and analyze all delta records when accessing an index page. As you can guess, a long chain of delta records affects performance. When this is the case, SQL Server consolidates delta records and rebuilds an index page, creating a new one. The newly created page has the same PID and replaces the old page, which is marked for garbage collection. Replacement of the page is accomplished by changing a pointer in the mapping table. SQL Server does not need to change internal pages because they use the mapping table to reference leaf-level pages.

The process of rebuilding is triggered at the moment a new delta record is created for pages that already have 16 delta records in a chain. The action described by the delta record, which triggers the rebuild, is incorporated into the newly created page.

Two other processes can create new or delete existing index pages, in addition to delta records consolidation. The first process, *page splitting*, occurs when a page does not have enough free space to accommodate a new data row. Another process, *page merging*, occurs when a delete operation leaves an index page less than 10% from the maximum page size, which is 8KB now, or when an index page contains just a single row.

■ **Note** The page splitting and page merging processes are covered in depth in Appendix B.

Obtaining Information About Nonclustered Indexes

In addition to the `sys.dm_db_xtp_hash_index_stats` view, which was discussed in Chapter 4, SQL Server provides two other views to obtain information about indexes on memory-optimized tables. Those views provide the data collected since the time when memory-optimized tables were loaded into memory, which occurs at database startup.

You can obtain information about index access methods and ghost rows in both hash and nonclustered indexes with the `sys.dm_db_xtp_index_stats` view. The notable columns in the view are the following:

- `scans_started` shows the number of times that row chains in the index were scanned. Due to the nature of the index, every operation, such as `SELECT`, `INSERT`, `UPDATE`, and `DELETE`, requires SQL Server to scan a row chain and increment this column.
- `rows_returned` represents the cumulative number of rows returned to the next operator in the execution plan. It does not necessarily match the number of rows returned to a client because further operators in the execution plan can change it.
- `rows_touched` represents the cumulative number of rows accessed in the index.

- `rows_expired` shows the number of detected stale rows. I will discuss this in greater detail when I talk about the garbage collection process in Chapter 9.
- `rows_expired_removed` returns the number of stale rows that have been unlinked from the index row chains. I will also discuss this in more detail when I talk about garbage collection.

Listing 5-7 shows the query that returns the information about indexes defined on the `dbo.Customers` table.

Listing 5-7. Querying the sys.dm_db_xtp_index_stats View

```
select
    s.name + '.' + t.name as [table]
    ,i.index_id
    ,i.name as [index]
    ,i.type_desc as [type]
    ,st.scans_started
    ,st.rows_returned
    ,iif(st.scans_started = 0, 0,
        floor(st.rows_returned / st.scans_started))
                as [rows per scan]
from
    sys.dm_db_xtp_index_stats st join sys.tables t on
        st.object_id = t.object_id
    join sys.indexes i on
        st.object_id = i.object_id and
        st.index_id = i.index_id
    join sys.schemas s on
        s.schema_id = t.schema_id
where
    s.name = 'dbo' and t.name = 'Customers'
```

Figure 5-6 illustrates the output of the query. Large number of `Rows Per Scan` can indicates heavy index scans, which can be the sign of a suboptimal indexing strategy and/or poorly written queries.

	Table	index_id	Index	Type	scans_started	rows_returned	Rows per Scan
1	dbo.Customers	4	PK_Customers	NONCLUSTERED HASH	570	5041	8
2	dbo.Customers	2	IDX_FullName	NONCLUSTERED	7	33	4
3	dbo.Customers	3	IDX_LastName_FirstName	NONCLUSTERED	9	684	76

Figure 5-6. *Output from the sys.dm_db_xtp_index_stats view*

■ **Note** You can read more about the `sys.dm_db_xtp_index_stats` view at http://msdn.microsoft.com/en-us/library/dn133081.aspx.

The `sys.dm_db_xtp_nonclustered_index_stats` view returns information about nonclustered indexes. It includes information about the total number of pages in the index plus page splits, merges, and consolidation-related statistics.

Listing 5-8 shows information about nonclustered indexes defined on the `dbo.Customers` table. Figure 5-7 shows the output of the query.

Listing 5-8. Querying the sys.dm_db_xtp_nonclustered_index_stats View

```
select
    s.name + '.' + t.name as [table]
    ,i.index_id
    ,i.name as [index]
    ,i.type_desc as [type]
    ,st.delta_pages
    ,st.leaf_pages
    ,st.internal_pages
    ,st.leaf_pages + st.delta_pages + st.internal_pages
                as [total pages]
from
    sys.dm_db_xtp_nonclustered_index_stats st
        join sys.tables t on
            st.object_id = t.object_id
        join sys.indexes i on
            st.object_id = i.object_id and
            st.index_id = i.index_id
        join sys.schemas s on
            s.schema_id = t.schema_id
where
    s.name = 'dbo' and t.name = 'Customers'
```

	Table	index_id	Index	Type	delta_pages	leaf_pages	internal_pages	Total Pages
1	dbo.Customers	2	IDX_FullName	NONCLUSTERED	64	5	1	70
2	dbo.Customers	3	IDX_LastName_FirstName	NONCLUSTERED	80	6	1	87

Figure 5-7. *Output from the sys.dm_db_xtp_nonclustered_index_stats view*

■ **Note** You can read more about the `sys.dm_db_xtp_nonclustered_index_stats` view at https://msdn.microsoft.com/en-us/library/dn645468.aspx.

Hash Indexes vs. Nonclustered Indexes

As you already know, hash indexes are useful only for point-lookup searches in cases when queries use equality predicates on all index columns. Nonclustered indexes, on the other hand, can be used in a much wider scope, which often makes the choice obvious. You should use nonclustered indexes when your queries benefit from scenarios other than point-lookups.

The situation is less obvious in the case of point-lookups. With the hash indexes, SQL Server can locate the hash bucket, which is the entry point to the data row chain, in a single step by calling the hash function and calculating the hash value. With nonclustered indexes, SQL Server has to traverse Bw-Tree to find a leaf page, and the number of steps depends on the height of the index and number of delta records there.

Even though nonclustered indexes require more steps to find an entry point to the data row chain, the chain can be smaller compared to hash indexes. Row chains in nonclustered indexes are built based on unique index key values. In hash indexes, row chains are built based on a non-unique hash key and can be larger due to hash collisions, especially when the `bucket_count` is insufficient.

Let's compare hash and nonclustered index performance in a point-lookup scenario. Listing 5-9 creates four tables of the same structure. Three of them have hash indexes defined on the `Value` column using a different `bucket_count`. The fourth table has a nonclustered index defined on the same column instead. Finally, the code populates all tables with the same data.

Listing 5-9. Hash and Nonclustered Indexes' Point Lookup Performance: Tables Creation

```
create table dbo.Hash_131072
(
    Id int not null
    constraint PK_Hash_131072
        primary key nonclustered
        hash with (bucket_count=131072),
    Value int not null,

    index IDX_Value hash(Value)
    with (bucket_count=131072)
)
with (memory_optimized=on, durability=schema_only);

create table dbo.Hash_16384
(
    Id int not null
        constraint PK_Hash_16384
        primary key nonclustered
        hash with (bucket_count=16384),
    Value int not null,
```

```
    index IDX_Value hash(Value)
    with (bucket_count=16384)
)
with (memory_optimized=on, durability=schema_only);

create table dbo.Hash_1024
(
    Id int not null
        constraint PK_Hash_1014
        primary key nonclustered
        hash with (bucket_count=1024),
    Value int not null,

    index IDX_Value hash(Value)
    with (bucket_count=1024)
)
with (memory_optimized=on, durability=schema_only);

create table dbo.NonClusteredIdx
(
    Id int not null
        constraint PK_NonClusteredIdx
        primary key nonclustered
        hash with (bucket_count=131072),
    Value int not null,

    index IDX_Value nonclustered(Value)
)
with (memory_optimized=on, durability=schema_only);
go

;with N1(C) as (select 0 union all select 0) -- 2 rows
,N2(C) as (select 0 from N1 as t1 cross join N1 as t2) -- 4 rows
,N3(C) as (select 0 from N2 as t1 cross join N2 as t2) -- 16 rows
,N4(C) as (select 0 from N3 as t1 cross join N3 as t2) -- 256 rows
,N5(C) as (select 0 from N4 as t1 cross join N4 as t2) -- 65,536 rows
,N6(C) as (select 0 from N5 as t1 cross join N1 as t2) -- 131,072 rows
,Ids(Id) as (select row_number() over (order by (select null)) from N6)
insert into dbo.Hash_131072(Id,Value)
    select Id, Id
    from ids
    where Id <= 75000;

insert into dbo.Hash_16384(Id,Value)
    select Id, Value
    from dbo.Hash_131072;
```

```
insert into dbo.Hash_1024(Id,Value)
    select Id, Value
    from dbo.Hash_131072;

insert into dbo.NonClusteredIdx(Id,Value)
    select Id, Value
    from dbo.Hash_131072;
```

Different numbers of buckets led to the different index row chain sizes in the indexes. In this case, the `Hash_131072`, `Hash_16384`, and `Hash_1024` tables have on average 1, 4, and 73 rows per chain, respectively.

■ **Tip** You can analyze hash index properties using the `sys.dm_db_xtp_hash_index_stats` view and the code from Listing 4-2 from Chapter 4.

As the next step, let's compare point-lookup performance using the code from Listing 5-10. This code triggers 65,536 point lookup selects against each table.

Listing 5-10. Hash and Nonclustered Indexes' Point Lookup Performance: Selecting Data

```
declare
    @T table(Value int not null primary key)

;with N1(C) as (select 0 union all select 0) -- 2 rows
,N2(C) as (select 0 from N1 as t1 cross join N1 as t2) -- 4 rows
,N3(C) as (select 0 from N2 as t1 cross join N2 as t2) -- 16 rows
,N4(C) as (select 0 from N3 as t1 cross join N3 as t2) -- 256 rows
,N5(C) as (select 0 from N4 as t1 cross join N4 as t2) -- 65,536 rows
,Ids(Id) as (select row_number() over (order by (select null)) from N5)
insert into @T(Value)
    select Id from Ids;

select t.Value, c.Cnt
from @T t
    cross apply
    (
       select count(*) as Cnt
       from dbo.Hash_131072 h
       where h.Value = t.Value
    ) c;
```

```
select t.Value, c.Cnt
from @T t
    cross apply
    (
       select count(*) as Cnt
       from dbo.Hash_16384 h
       where h.Value = t.Value
    ) c;

select t.Value, c.Cnt
from @T t
    cross apply
    (
       select count(*) as Cnt
       from dbo.Hash_1024 h
       where h.Value = t.Value
    ) c;

select t.Value, c.Cnt
from @T t
    cross apply
    (
       select count(*) as Cnt
       from dbo.NonClusteredIdx h
       where h.Value = t.Value
    ) c;
```

Table 5-1 shows the execution time of the queries in my environment. As you can see, the nonclustered index point-lookup select is slightly slower compared to hash indexes with relatively short row chains; however, it is faster in the case of the long row chains and incorrect bucket count estimations.

Table 5-1. *Execution Time of Queries*

	Hash_131072	Hash_16384	Hash_1024	NonClusteredIdx
Average Index Row Chain Size	1	4	73	N/A
Execution Time	141 ms	156 ms	219 ms	171 ms

Memory requirements are another factor to consider. With the hash indexes, memory usage depends on the number of buckets. The amount of memory required for the nonclustered indexes depends on the size of the index key and index cardinality (uniqueness of index key values). For example, if a table has a `varchar` column with 1,000,000 unique values of 100 bytes each, the nonclustered index on that column would require about 800MB to support a Bw-Tree structure. Alternatively, a hash index with 2,097,152 buckets will use just 16MB of memory.

With all that being said, hash indexes are a good choice only in cases where the workload and data are relatively static and, therefore, you can correctly estimate `bucket_count` and you do not expect anything other than point-lookup queries in the future. In all other cases, nonclustered indexes are the safer choice.

Summary

Nonclustered indexes are the second type of indexes supported by the In-Memory OLTP Engine. They have similar SARGability rules with the B-Tree indexes defined on on-disk tables with the exception of the scans in the order opposite of the index key sorting order.

Internally, nonclustered indexes use a lock- and latch-free variation of B-Tree, called Bw-Tree, which consists of internal and leaf data pages referencing each other through the mapping table. Leaf data pages store one row per each individual key value with the pointer to the chain of the data rows with the same key.

SQL Server never updates index pages. Any changes are referenced through the delta records that correspond to individual `INSERT` and `DELETE` operations on the page. SQL Server consolidates the large chains of delta records and performs splitting and merging of the data pages when needed. All of those processes create the new data pages, marking the old ones for garbage collection.

Nonclustered indexes are a good choice in scenarios when point-lookup is not an option and/or when it is hard to estimate the number of buckets in the hash index.

CHAPTER 6

In-Memory OLTP Programmability

This chapter focuses on the programmability aspects of the In-Memory OLTP Engine in SQL Server. It describes the process of native compilation, and it provides an overview of the natively compiled stored procedures and T-SQL features that are supported in In-Memory OLTP. Finally, this chapter compares the performance of several use cases that access and modify data in memory-optimized tables using natively compiled stored procedures and interpreted T-SQL with the interop engine.

Native Compilation

As you already know, memory-optimized tables can be accessed from regular T-SQL code using the query interop engine. This approach is very flexible. As long as you work within the supported feature set, the location of the data is transparent. The code does not need to know, nor does it need to worry about, if it works with on-disk or with memory-optimized tables.

Unfortunately, this flexibility comes at a cost. T-SQL is an interpreted and CPU-intensive language. Even a simple T-SQL statement requires thousands, and sometimes millions, of CPU instructions to execute. Even though the in-memory data location speeds up data access and eliminates latching and locking contentions, the overhead of T-SQL interpretation sets limits on the level of performance improvements achievable with In-Memory OLTP.

In practice, it is common to see a 2X-4X system throughput increase when memory-optimized data is accessed through the interop engine. To improve performance even further, In-Memory OLTP utilizes native compilation. As a first step, it converts any row-data manipulation and access logic into C code, which is compiled into DLLs and loaded into SQL Server's process memory. These DLLs (one per table) consist of native CPU instructions, and they execute without any further code interpretation overhead of T-SQL statements.

Consider the simple situation where you need to read the value of a fixed-length column from a data row. In the case of on-disk tables, SQL Server obtains the starting offset and length of the column from the system catalogs, and it performs the required manipulations to convert the sequence of bytes to the required data type.

With memory-optimized tables, the DLL already knows the column offset and data type. SQL Server can read data from a pre-defined offset in a row using a pointer of the correct data type without any further overhead involved. As you can guess, this approach dramatically reduces the number of CPU instructions required for the operation.

On the flip side, this approach brings some limitations. You cannot change the format of a row after the DLL is generated. The compiled code would not know anything about the changes. This problem is more complicated than it seems, and a simple recompilation of the DLL does not address it.

Again, consider the situation where you need to add another nullable column to a table. This is a metadata-level operation for on-disk tables, which does not change the data in existing table rows. T-SQL would be able to detect that column data is not present by analyzing the various data row properties at runtime.

The situation is far more complicated in the case of memory-optimized tables and natively compiled code. It is easy to generate a new version of the DLL that knows about the new data column; however, that is not enough. The DLL needs to handle different versions of rows and different data formats depending on the presence of column data. While this is technically possible, it adds extra logic to the DLL, which leads to additional processing instructions, which slows data access. Moreover, the logic to support multiple data formats remains in the code forever, degrading performance even further with each table alteration.

While, technically speaking, it is possible to convert all existing data rows to the new format, this operation requires exclusive access to the table, which violates In-Memory OLTP lock- and latch-free principles and is not supported in SQL Server 2014.

■ **Tip** The only way to alter a table and change its schema and index definition is to drop and recreate the table, staging data somewhere during the process. This was discussed in detail in Chapter 4.

To reduce the overhead of the T-SQL interpretation even further, the In-Memory OLTP Engine allows you to perform native compilation of the stored procedures. These stored procedures are compiled in the same way as table-related DLLs and are also loaded to the SQL Server process memory.

Native compilation utilizes both the SQL Server and In-Memory OLTP Engines. As a first step, SQL Server parses the T-SQL code and, in the case of stored procedures, it generates an execution plan using the Query Optimizer. At the end of this stage, SQL Server generates a structure called *MAT (Mixed Abstract Tree)*, which represents metadata, imperative logic, expressions, and query plans. I will discuss how SQL Server optimizes natively compiled stored procedures later in this chapter.

As a next step, In-Memory OLTP transforms MAT to another structure called *PIT (Pure Imperative Tree),* which is used to generate source code that is compiled and linked into the DLL.

Figure 6-1 illustrates the process of native compilation in SQL Server.

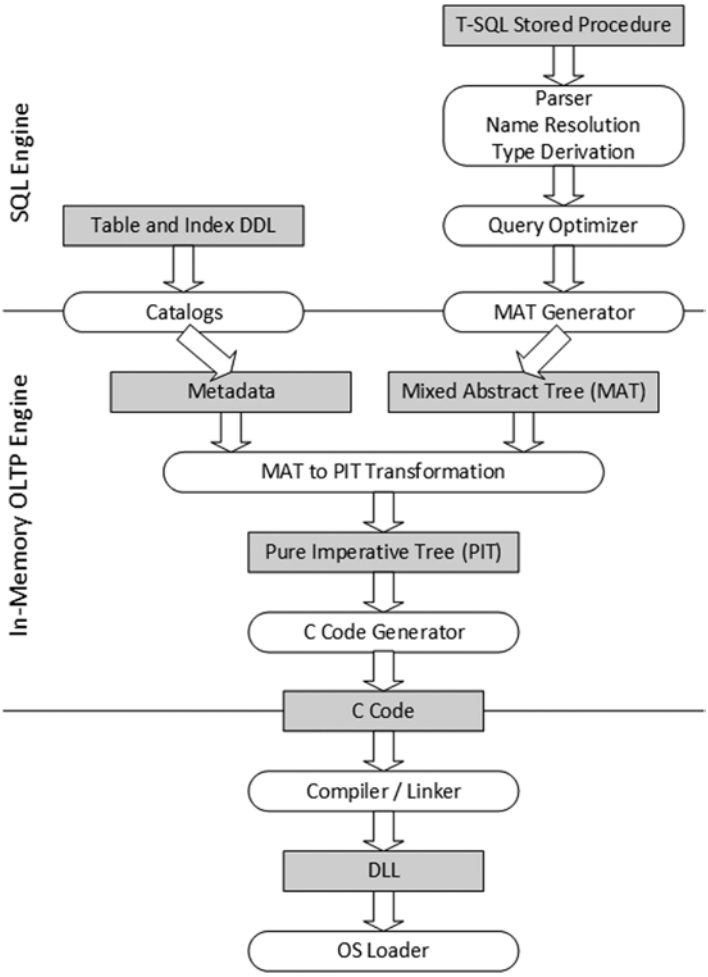


Figure 6-1. *Native compilation in SQL Server*

The code generated for native compilation uses the plain C language and is very efficient. It is very hard to read, however. For example, every method is implemented as a single function, which does not call other functions but rather implements its code inline using `GOTO` as a control flow statement. The intention has never been to generate human-readable code; it is used as the source for native compilation only.

Binary DLL files are not persisted in a database backup. SQL Server recreates table-related DLLs on database startup and stored procedures-related DLLs at the time of the first call. This approach mitigates security risks from hackers, who can substitute DLLs with malicious copies. It is important to remember this behavior because it can add overhead at database startup time and change the execution plans of natively compiled stored procedures after a database restart.

■ **Tip** Natively compiled stored procedures are usually faster than interpreted T-SQL ones. However, their compilation time can be significantly longer compared to T-SQL stored procedures. You should remember this behavior and avoid using extremely short timeouts in natively compiled stored procedure calls.

SQL Server places binary DLLs and all other native compilation-related files in an XTP subfolder under the main SQL Server data directory. It groups files on a per-database basis by creating another level of subfolders. Figure 6-2 shows the content of the folder for the database (with ID=5), which contains the memory-optimized tables and a natively compiled stored procedures you created in previous chapters of this book.

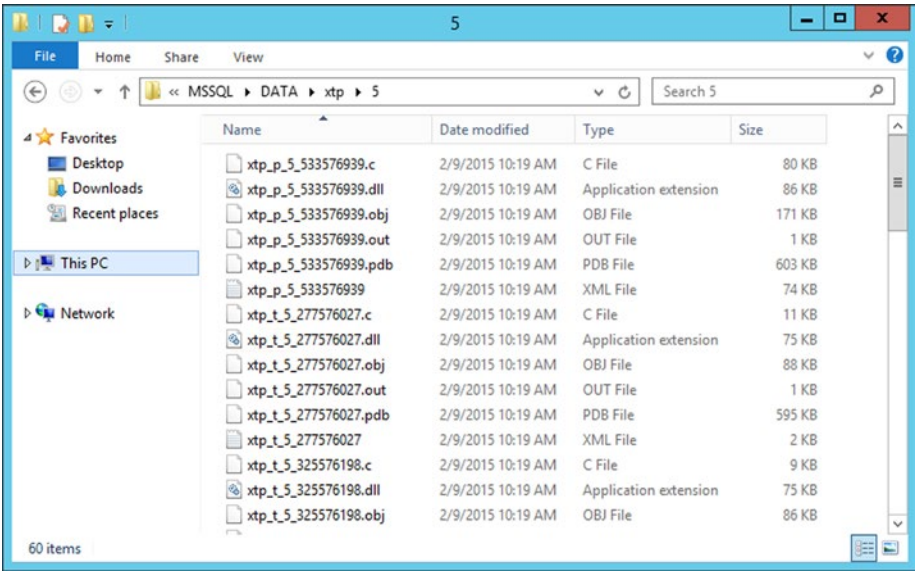


Figure 6-2. *Folder with natively compiled objects*

All of the file names start with the prefix `xtp_` followed either by the `p` (stored procedure) or `t` (table) character, which indicates the object type. The two last parts of the name include the database and object IDs for the object.

File extensions determine the type of the file, such as:

- `*.mat.xml` files store an XML representation of the MAT structure.
- `*.c` files are the source file generated by the C code generator.
- `*.obj` are the object files generated by the C compiler.
- `*.pub` are symbol files produced by the C compiler.

- `*.out` are log files from the C compiler.
- `*.dll` are natively compiled DLLs generated by the C linker. Those files are loaded into SQL Server memory and used by the In-Memory OLTP Engine.

■ **Tip** You can open and analyze the C source code and XML MAT in the text editor application to get a sense of the native compilation process.

Listing 6-1 shows how to obtain a list of the natively compiled objects loaded into SQL Server memory. It also returns the list of tables and stored procedures from the database to show the correlation between a DLL file name and object IDs.

Listing 6-1. Obtaining a List of Natively Compiled Objects Loaded into SQL Server Memory

```
select
    s.name + '.' + o.name as [Object Name]
    ,o.object_id
from
     (
        select schema_id, name, object_id
        from sys.tables
        where is_memory_optimized = 1
        union all
        select schema_id, name, object_id
        from sys.procedures
    ) o join sys.schemas s on
        o.schema_id = s.schema_id;

select base_address, file_version, language, description, name
from sys.dm_os_loaded_modules
where description = 'XTP Native DLL';
```

Figure 6-3 illustrates the output of the code.

	Object Name	object_id
1	dbo.WebRequests_Memory	277576027
2	dbo.WebRequestHeaders_Memory	325576198
3	dbo.WebRequestParams_Memory	357576312
4	dbo.HashIndex_HighBucketCount	1301579675

	base_address	file_version	language	description	name
1	0x00007FFF6CE10000	0.0:0.0	67699940	XTP Native DLL	M:\DATA\xtp\5\xtp_t_5_277576027.dll
2	0x00007FFF6C380000	0.0:0.0	67699940	XTP Native DLL	M:\DATA\xtp\5\xtp_t_5_325576198.dll
3	0x00007FFF6BEE0000	0.0:0.0	67699940	XTP Native DLL	M:\DATA\xtp\5\xtp_t_5_357576312.dll
4	0x00007FFF6B990000	0.0:0.0	67699940	XTP Native DLL	M:\DATA\xtp\5\xtp_p_5_533576939.dll
5	0x00007FFF67A80000	0.0:0.0	67699940	XTP Native DLL	M:\DATA\xtp\5\xtp_t_5_1269579561...
6	0x00007FFF58440000	0.0:0.0	67699940	XTP Native DLL	M:\DATA\xtp\5\xtp_t_5_1301579675...
7	0x00007FFF692F0000	0.0:0.0	67699940	XTP Native DLL	M:\DATA\xtp\5\xtp_t_5_1365579903...
8	0x00007FFF69330000	0.0:0.0	67699940	XTP Native DLL	M:\DATA\xtp\5\xtp_t_5_1397580017...

Figure 6-3. *Natively compiled objects loaded into SQL Server memory*

Natively Compiled Stored Procedures

Natively compiled stored procedures are the stored procedures that are compiled into native code. They are extremely efficient, and they can provide major performance improvements when working with memory-optimized tables, compared to interpreted T-SQL statements, which access those tables through the query interop component.

■ **Note** In this chapter, I will reference regular interpreted (non-natively compiled) stored procedures as *T-SQL procedures*.

Creating Natively Compiled Stored Procedures

As you already know, you can create natively compiled stored procedures using the regular `CREATE PROCEDURE` statement and T-SQL language. However, those procedures have several additional options that need to be specified. Listing 6-2 shows the structure of natively compiled stored procedures along with those options.

Listing 6-2. Natively Compiled Stored Procedure Structure

```
create proc dbo.NativelyCompiledProc
(
    /* Parameters */
    @Param1 int not null = 1
    ,@Param2 int
)
```

```
with
    native_compilation    -- Indicates natively compiled SP
    ,schemabinding    -- Required
    ,execute as owner    -- execute as OWNER/SELF/USER is required
as
-- Natively compiled SPs are executed as atomic blocks (all or nothing)
begin atomic with
(
    transaction isolation level = snapshot    -- Isolation Level is required
    ,language = N'English'    -- Required langugage setting for SP
    ,delayed_durability = off    -- Optional
    ,datefirst = 7    -- Optional
    ,dateformat = 'mdy'    -- Optional
)
    /* Stored Procedure Body */
end
```

You can define parameters of natively compiled stored procedures the same way as with T-SQL procedures. However, natively compiled stored procedures allow you to specify if parameters are required and must be provided at the time of a call using the `NOT NULL` construct in the definition. SQL Server raises an error if you do not provide their values at the time of the call.

■ **Important** It is recommended that you avoid type conversion and do not use named parameters when you call natively compiled stored procedures. It is more efficient to use the `exec Proc value [,..,value]` rather than the `exec Proc @Param=value [,..,@Param=value]` calling format.

You can detect inefficient parameterization with the `hekaton_slow_parameter_parsing` extended event.

All natively compiled stored procedures must be schema bound and have the security context `EXECUTE AS OWNER/SELF/USER` specified. The default `EXECUTE AS CALLER` context is not supported to avoid the overhead of per-statement permission checks during the execution.

Two other required options include the transaction isolation level and the language setting, which controls a message's language and default date format. Natively compiled stored procedures do not use the runtime `SET LANGUAGE` session option, relying on the `LANGUAGE` setting instead.

You can control date format, first day of the week, and delayed durability of a stored procedure using the `DATEFORMAT`, `DATEFIRST`, and `DELAYED_DURABILITY` settings, respectively.

■ **Note** Delayed durability is a SQL Server 2014 feature that controls how SQL Server hardens log records, flushing them from the log buffer to the transaction log. Enabling delayed durability can help to improve transaction throughput in very busy OLTP systems at the cost of a possible small data loss in the event of an unexpected SQL Server shutdown or crash.

You can read more about delayed durability at https://msdn.microsoft.com/en-us/library/dn449490.aspx. You can also read about it in Chapter 29 of my *Pro SQL Server Internals* book.

Natively compiled stored procedures are executed as the atomic blocks, which is *all or nothing* approach; either all statements in the procedure succeed or all of them fail. I will discuss how atomic blocks work later in the chapter.

As mentioned, you can define the natively compiled stored procedure body pretty much the same way as regular T-SQL procedures. However, the natively compiled stored procedures support only a limited set of T-SQL constructs. Let's look at the supported features and limitations in different T-SQL areas in detail.

Supported T-SQL Features

One of the biggest limitations of natively compiled stored procedures is that they can access only memory-optimized tables. The only option to join data from memory-optimized and on-disk tables is to use the interpreted T-SQL and interop engine.

The following T-SQL features and constructs are supported and can be used in natively compiled stored procedures.

Control Flow

The following control flow options are supported:

- `IF` and `WHILE`.
- Assigning a value to a variable with the `SELECT` and `SET` operators.
- `RETURN`.
- `TRY/CATCH/THROW` (`RAISERROR` is not supported). It is recommended that you use a single `TRY/CATCH` block for the entire stored procedure for better performance.
- It is possible to declare variables as `NOT NULL` as long as they have an initializer as part of the `DECLARE` statement.

Query Surface Area

The following query surface area functions are supported:

- SELECT, INSERT, UPDATE, and DELETE. However, you cannot use multiple VALUE clauses with the single INSERT statement.
- CROSS JOIN and INNER JOIN are the only join types supported. Moreover, you can use joins only with SELECT operators.
- Expressions in the SELECT list and the WHERE and HAVING clauses are supported as long as they use supported operators.
- IS NULL and IS NOT NULL.
- GROUP BY is supported with the exception of grouping by string or binary data.
- TOP and ORDER BY. However, you cannot use WITH TIES and PERCENT in the TOP clause. Moreover, the TOP operator is limited to 8,192 rows when the TOP <constant> is used, or even a lesser number of rows in the case of joins. You can address this last limitation by using a TOP <variable> approach. However, it is less efficient in terms of performance.
- INDEX, FORCESCAN, FORCESEEK, FORCE ORDER, INNER LOOP JOIN, and OPTIMIZE FOR hints.

Note that the DISTINCT operator is **not** supported.

Operators

The following operators are supported:

- Comparison operators, such as =, <, <=, >, >=, <> and BETWEEN.
- Unary and binary operators, such as +, -, *, /, %. Note that + operators are supported for both numbers and strings.
- Bitwise operators, such as &, |, ~, ^.
- Logical operators, such as AND, OR, and NOT. However, the OR and NOT operators are not supported in the WHERE and HAVING clauses of the query.

Build-In Functions

The following build-in functions are supported:

- Math functions: `ACOS`, `ASIN`, `ATAN`, `ATN2`, `COS`, `COT`, `DEGREES`, `EXP`, `LOG`, `LOG10`, `PI`, `POWER`, `RAND`, `SIN`, `SQRT`, `SQUARE`, and `TAN`.
- Date/time functions: `CURRENT_TIMESTAMP`, `DATEADD`, `DATEDIFF`, `DATEFROMPARTS`, `DATEPART`, `DATETIME2FROMPARTS`, `DATETIMEFROMPARTS`, `DAY`, `EOMONTH`, `GETDATE`, `GETUTCDATE`, `MONTH`, `SMALLDATETIMEFROMPARTS`, `SYSDATETIME`, `SYSUTCDATETIME`, and `YEAR`.
- String functions: `LEN`, `LTRIM`, `RTRIM`, and `SUBSTRING`.
- Error functions: `ERROR_LINE`, `ERROR_MESSAGE`, `ERROR_NUMBER`, `ERROR_PROCEDURE`, `ERROR_SEVERITY`, and `ERROR_STATE`.
- `NEWID` and `NEWSEQUENTIALID`.
- `CAST` and `CONVERT`. However, it is impossible to convert between a non-Unicode and a Unicode string.
- `ISNULL`.
- `SCOPE_IDENTITY`.
- You can use `@@ROWCOUNT` within a natively-compiled stored procedure; however, its value is reset to 0 at the beginning and end of the procedure.

Atomic Blocks

Natively compiled stored procedures execute as atomic blocks, which is an *all or nothing* approach; either all statements in the procedure succeed or all of them fail.

When a natively compiled stored procedure is called outside of the context of an active transaction, it starts a new transaction and either commits or rolls it back at the end of the execution.

In cases where a procedure is called in the context of an active transaction, SQL Server creates a savepoint at the beginning of the procedure's execution. In case of an error in the procedure, SQL Server rolls back the transaction to the created savepoint. Based on the severity and type of the error, the transaction is either going to be able to continue and commit or become doomed and uncommittable.

Let's create a memory-optimized table and natively compiled stored procedure, as shown in Listing 6-3.

Listing 6-3. Atomic Blocks and Transactions: Object Creation

```
create table dbo.MOData
(
    ID int not null
        primary key nonclustered
        hash with (bucket_count=10),
    Value int null
)
with (memory_optimized=on, durability=schema_only);

insert into dbo.MOData(ID, Value)
values(1,1), (2,2)
go

create proc dbo.AtomicBlockDemo
(
    @ID1 int not null
    ,@Value1 bigint not null
    ,@ID2 int
    ,@Value2 bigint
)
with native_compilation, schemabinding, execute as owner
as
begin atomic
with (transaction isolation level = snapshot, language=N'English')

    update dbo.MOData set Value = @Value1 where ID = @ID1;

    if @ID2 is not null
        update dbo.MOData set Value = @Value2 where ID = @ID2;
end;
```

At this point, the `MOData` table has two rows with values `(1,1)` and `(2,2)`. As a first step, let's start the transaction and call a stored procedure twice, as shown in Listing 6-4.

Listing 6-4. Atomic Blocks and Transactions: Calling a Stored Procedure

```
begin tran
    exec dbo.AtomicBlockDemo 1, -1, 2, -2
    exec dbo.AtomicBlockDemo 1, 0, 2, 999999999999999
```

The first call of the stored procedure succeeds, while the second call triggers an arithmetic overflow error as shown:

```
Msg 8115, Level 16, State 0, Procedure AtomicBlockDemo, Line 49

Arithmetic overflow error converting bigint to data type int.
```

You can check that the transaction is still active and committable with this select: `SELECT @@TRANCOUNT as [@@TRANCOUNT], XACT_STATE() as [XACT_STATE()]`. It returns the following results:

```
@@TRANCOUNT    XACT_STATE()
-----------    ------------
1              1
```

If you commit the transaction and check the content of the table, you will see that the data reflects the changes caused by the first stored procedure call. Even though the first update statement from the second call succeeded, SQL Server rolled it back because the natively compiled stored procedure executed as an atomic block. You can see the data in the `MOData` table:

```
ID          Value
----------- -----------
1           -1
2           -2
```

As a second example, let's trigger a critical error, which dooms the transaction, making it uncommittable. One such situation is a write/write conflict, when multiple sessions are trying to update the same rows. You can trigger it by executing the code in Listing 6-5 in two different sessions.

■ **Note** I will talk about write/write conflicts and the In-Memory OLTP concurrency model in Chapter 7.

Listing 6-5. Atomic Blocks and Transactions: Write/Write Conflict

```
begin tran
    exec dbo.AtomicBlockDemo 1, 0, null, null
```

When you run the code in the second session, it triggers the following exception:

```
Msg 41302, Level 16, State 110, Procedure AtomicBlockDemo, Line 13

The current transaction attempted to update a record that has been updated
since this transaction started. The transaction was aborted.

Msg 3998, Level 16, State 1, Line 1

Uncommittable transaction is detected at the end of the batch. The
transaction is rolled back.
```

If you check `@@TRANCOUNT` in the second session, you will see that SQL Server terminates the transaction.

```
@@TRANCOUNT
-----------
0
```

Finally, it is worth mentioning that atomic blocks are an In-Memory OLTP feature and are not supported in T-SQL stored procedures.

Optimization of Natively Compiled Stored Procedures

Interpreted T-SQL stored procedures are compiled at the time of first execution. Additionally, they can be recompiled after they are evicted from plan cache and in a few other cases, such as outdated statistics, changes in database schema, or recompilation, which are explicitly requested in the code.

This behavior is different from natively compiled stored procedures, which are compiled at creation time. They are never recompiled, only with the exception of SQL Server or a database restart. In these cases, recompilation occurs at the time of the first stored procedure call.

SQL Server does not sniff parameters at the time of compilation, optimizing statements for `UNKNOWN` values. It uses memory optimized table statistics during optimization. However, as you already know, these statistics are not updated automatically, and they can be outdated at that time.

Fortunately, cardinality estimation errors have a smaller impact on performance in the case of natively compiled stored procedures. Contrary to on-disk tables, where such errors can lead to highly inefficient plans due to an incorrect index choice and, therefore, a high number of *Key* or *RID Lookup* operations, all indexes in memory-optimized tables reference the same data row and, in a nutshell, are covering indexes. Moreover, errors will not affect the choice of join strategy—the *inner nested loop* is the only physical join type supported in natively compiled stored procedures in the first release of In-Memory OLTP.

Outdated statistics at the time of compilation, however, can still lead to inefficient plans. One such example is a query with multiple predicates on indexed columns. SQL Server needs to know the index's selectivity to choose the most efficient one. Another example is the incorrect choice of inner and outer input for the nested loop join, which you saw in Chapter 4.

It is better to recompile natively compiled stored procedures if the data in the table has significantly changed. You can do it with the following actions:

1. Update the statistics to reflect the current data distribution in the table(s).
2. Script permissions assigned to natively compiled stored procedures.
3. Drop and recreate procedures. These actions force recompilation.
4. Assign required permissions to the procedures.

Finally, it is worth mentioning that the presence of natively compiled stored procedures requires you to adjust the deployment process in the system. It is common to create all database schema objects, including tables and stored procedures, at the beginning of deployment. While the time of deployment does not matter for T-SQL procedures, such a strategy compiles natively compiled stored procedures at a time when database tables are empty. You should recompile (recreate) natively compiled procedures later, after the tables are populated with data and statistics are up to date.

Interpreted T-SQL and Memory-Optimized Tables

The query interop component provides transparent, memory-optimized table access to interpreted T-SQL code. In the interpreted mode, SQL Server treats memory-optimized tables pretty much the same way as on-disk tables. It optimizes queries and caches execution plans, regardless of where the table is located. The same set of operators is used during query execution. From a high level, when the operator's `GetRow()` method is called, it is routed either to the Storage Engine or to the In-Memory OLTP Engine, depending on the underlying table type.

Most T-SQL features are supported in interpreted mode. There are still a few exceptions, however:

- `TRUNCATE TABLE`.
- `MERGE` operator with memory-optimized table as the target.
- Context connection from CLR code.
- Referencing memory-optimized tables in indexed views. You can reference memory-optimized tables in partitioned views, combining data from memory-optimized and on-disk tables.
- `DYNAMIC` and `KEYSET` cursors, which are automatically downgraded to `STATIC`.
- Cross-database queries and transactions.
- Linked servers.

As you can see, the list of limitations is pretty small. However, the flexibility of query interop access comes at a cost. Natively compiled stored procedures are usually more efficient compared to their interpreted T-SQL counterparts. In some cases, such as joins between memory-optimized and on-disk tables, query interop is the only choice; however, it is usually preferable to use natively compiled stored procedures when possible.

Performance Comparison

Let's run several tests comparing performance of several use cases that work with memory-optimized tables using natively compiled stored procedures and the interop engine.

Let's create two memory-optimized tables using a `schema_only` durability option to avoid any I/O and transaction logging overhead during the tests. You can see the code in Listing 6-6, which also creates a numbers' table and populates it with the values.

Listing 6-6. Native Compilation and Interop Mode Performance Comparison: Creating Test Tables

```
create table dbo.Customers
(
    CustomerId int not null
        primary key nonclustered
        hash with (bucket_count=200000),
    Name nvarchar(255)
        collate Latin1_General_100_BIN2 not null,
    CreatedOn datetime2(0) not null
        constraint DEF_Customers_CreatedOn
        default sysutcdatetime(),
    Placeholder char(200) not null,

    index IDX_Name nonclustered(Name)
)
with (memory_optimized=on, durability=schema_only);

create table dbo.Orders
(
    OrderId int not null
        primary key nonclustered
        hash with (bucket_count=5000000),
    CustomerId int not null,
    OrderNum varchar(32)
        collate Latin1_General_100_BIN2 not null,
    OrderDate datetime2(0) not null
        constraint DEF_Orders_OrderDate
        default sysutcdatetime(),
    Amount money not null,
    Placeholder char(200) not null,

    index IDX_CustomerId
    nonclustered hash(CustomerId)
    with (bucket_count=200000),

    index IDX_OrderNum nonclustered(OrderNum)
)
with (memory_optimized=on, durability=schema_only);
```

```
create table dbo.Numbers
(
    Num int not null
        Constraint PK_Numbers
        primary key clustered
);

;with N1(C) as (select 0 union all select 0) -- 2 rows
,N2(C) as (select 0 from N1 as t1 cross join N1 as t2) -- 4 rows
,N3(C) as (select 0 from N2 as t1 cross join N2 as t2) -- 16 rows
,N4(C) as (select 0 from N3 as t1 cross join N3 as t2) -- 256 rows
,N5(C) as (select 0 from N4 as t1 cross join N4 as t2) -- 65,536 rows
,N6(C) as (select 0 from N5 as t1 cross join N3 as t2) -- 1,048,576 rows
,Ids(Id) as (select row_number() over (order by (select null)) from N6)
insert into dbo.Numbers(Num)
    select Id from Ids;
```

As the first step, we will measure INSERT performance using three different approaches and batches of different sizes. The first two stored procedures, InsertCustomers_Row and InsertCustomers_NativelyCompiled, will run INSERT statements on per-row basis using the interop engine and native compilation, respectively. The third stored procedure, InsertCustomers_Batch, will insert all rows in the single batch through the interop engine. Listing 6-7 shows the implementation of the stored procedures.

Listing 6-7. Native Compilation and Interop Mode Performance Comparison: Inserting Data into the dbo.Customers Table

```
create proc dbo.InsertCustomers_Row
(
    @NumCustomers int
)
as
begin
    set nocount on
    set xact_abort on

    declare
        @I int = 1;

    begin tran
        while @I <= @NumCustomers
        begin
            insert into dbo.Customers(CustomerId,Name,Placeholder)
            values(@I,N'Customer ' + convert(nvarchar(10),@I),'Data');
```

```
            set @I += 1;
        end;
    commit
end
go

create proc dbo.InsertCustomers_Batch
(
    @NumCustomers int
)
as
begin
    set nocount on
    set xact_abort on

    if @NumCustomers > 1048576
    begin
        raiserror('@NumCustomers should not exceed 1,048,576',10,1);
        return;
    end;

    begin tran
        insert into dbo.Customers(CustomerId,Name,Placeholder)
            select Num, N'Customer ' + convert(nvarchar(10),Num),'Data'
            from dbo.Numbers
            where Num <= @NumCustomers
    commit
end
go

create proc dbo.InsertCustomers_NativelyCompiled
(
    @NumCustomers int not null
)
with native_compilation, schemabinding, execute as owner
as
begin atomic with
(
    transaction isolation level = snapshot
    ,language = N'English'
)
    declare
        @I int = 1;
```

```
    while @I <= @NumCustomers
    begin
        insert into dbo.Customers(CustomerId,Name,Placeholder)
        values(@I,N'Customer ' + convert(nvarchar(10),@I), 'Data');

        set @I += 1;
    end;
end;
```

Table 6-1 shows the execution time of each stored procedure for the batches of 10,000, 50,000, and 100,000 rows in my environment. As you can see, the natively compiled stored procedure is about four times faster at row-by-row inserts and about 30-40 percent faster even compared to batch inserts through the interop engine.

Table 6-1. *Execution Times of InsertCustomers Stored Procedures*

	10,000 rows	50,000 rows	100,000 rows
InsertCustomers_Row	220ms	1,160ms	2,170ms
InsertCustomers_Batch	98ms	446ms	886ms
InsertCustomers_NativelyCompiled	60ms	270ms	533ms

As the next step, let's compare performance of `UPDATE` operations. Listing 6-8 shows a natively compiled stored procedure that updates 50 percent of the rows in the `Customers` table.

Listing 6-8. Native Compilation and Interop Mode Performance Comparison: Natively Compiled Stored Procedure That Updates Data in the dbo.Customers Table

```
create proc dbo.UpdateCustomers
(
    @Placeholder char(100) not null
)
with native_compilation, schemabinding, execute as owner
as
begin atomic with
(
    transaction isolation level = snapshot
    ,language = N'English'
)
    update dbo.Customers
    set Placeholder = @Placeholder
    where CustomerId % 2 = 0;
end;
```

Table 6-2 shows the execution time of the UpdateCustomers stored procedure and the same UPDATE statement executed through the interop engine. As you see, the natively compiled stored procedure is about three times faster than the interop approach.

Table 6-2. *Execution Times of Update Operations*

dbo.UpdateCustomers Natively Compiled Stored Procedure	UPDATE Statement Executed Through Interop Engine
113ms	380 ms

In the next step, let's compare the performance of a SELECT query that joins data from the Customers and Orders tables and performs sorting and aggregations. I have populated the Orders table with 1,000,000 rows evenly distributed between 100,000 customers before the test. Listing 6-9 shows the natively compiled stored procedure with the query.

Listing 6-9. Native Compilation and Interop Mode Performance Comparison: Natively Compiled Stored Procedure with SELECT Query

```
create proc dbo.GetTopCustomers
with native_compilation, schemabinding, execute as owner
as
begin atomic with
(
    transaction isolation level = snapshot
    ,language = N'English'
)
    select top 10
        c.CustomerId, c.Name, count(o.OrderId) as [Order Cnt]
        ,max(o.OrderDate) as [Most Recent Order Date]
        ,sum(o.Amount) as [Total Amount]
    from
        dbo.Customers c join dbo.Orders o on
            c.CustomerId = o.CustomerId
    group by
        c.CustomerId, c.Name
    order by
        sum(o.Amount) desc;
end;
```

Table 6-3 shows the execution times of the stored procedure and the same query executed through the interop engine. As you see, the natively compiled stored procedure is about eight times faster in this scenario.

Table 6-3. *Execution Times of Select Operations*

dbo.GetTopCustomers Natively Compiled Stored Procedure	SELECT Statement Executed Through Interop Engine
366ms	2,763 ms

It is very important to remember, however, that natively compiled stored procedures do not support hash and merge joins, which could outperform nested loop joins on large and unsorted inputs.

Finally, let's compare the performance of `DELETE` operations. Listing 6-10 shows a natively compiled stored procedure that deletes the data from both tables.

Listing 6-10. Native Compilation and Interop Mode Performance Comparison: Natively Compiled Stored Procedure That Deletes the Data from Both Tables

```
create proc dbo.DeleteCustomersAndOrders
with native_compilation, schemabinding, execute as owner
as
begin atomic with
(
    transaction isolation level = snapshot
    ,language = N'English'
)
    delete from dbo.Orders;
    delete from dbo.Customers;
end;
```

Table 6-4 shows the execution times of the stored procedure and `DELETE` statements executed through the interop engine. In both cases, the `Customers` and `Orders` tables were populated with the same data, which is 100,000 and 1,000,000 rows respectively. Again, the natively compiled stored procedure is faster.

Table 6-4. *Execution Times of Delete Operations*

dbo.DeleteCustomersAndOrders Natively Compiled Stored Procedure	DELETE Statements Executed Through Interop Engine
1,053 ms	1,640 ms

As you have seen, native compilation provides significant performance improvements compared to the interop engine. It is beneficial to use it as long as the limitations do not prevent you from implementing the logic, and the additional administration and maintenance overhead is acceptable.

Lastly, you should remember that SQL Server 2014 does not support parallel execution plans for the statements that access memory-optimized tables. It makes In-Memory OLTP the bad candidate for Data Warehouse workload with the large scans and complex aggregations. We will discuss those scenarious in greater details in Chapter 11.

Memory-Optimized Table Types and Variables

SQL Server allows you to create memory-optimized table types. Table variables of these types are called *memory-optimized table variables*. In contrast to regular disk-based table variables, memory-optimized table variables live in memory only and do not utilize `tempdb`.

Memory-optimized table variables provide great performance. They can be used as a replacement for disk-based table variables and, in some cases, temporary tables. Obviously, they have the same set of functional limitations as memory-optimized tables.

Contrary to disk-based table types, you can define indexes on memory-optimized table types. The same statistics-related limitations still apply; however, as discussed, due to the nature of indexes on memory-optimized tables, cardinality estimation errors yield a much lower negative impact compared to those of on-disk tables.

■ **Important** As the opposite of on-disk table variables, statement-level recompile does not allow Query Optimizer to obtain the number of rows in memory-optimized table variables. It always estimates that memory-optimized table variables have just a single row.

SQL Server does not support inline declaration of memory-optimized table variables. For example, the code shown in Listing 6-11 will not compile and it will raise an error. The reason behind this limitation is that SQL Server compiles a DLL for every memory-optimized table type, which will not work in the case of inline declaration.

Listing 6-11. (Non-functional) Inline Declaration of Memory-Optimized Table Variables

```
declare
    @IDList table
     (
        ID int not null
            primary key nonclustered hash
            with (bucket_count=10000)
    )
    with (memory_optimized=on)
```

```
Msg 319, Level 15, State 1, Line 91

Incorrect syntax near the keyword 'with'. If this statement is a common
table expression, an xmlnamespaces clause or a change tracking context
clause, the previous statement must be terminated with a semicolon.
```

You should define and use a memory-optimized table type instead, as shown in Listing 6-12.

Listing 6-12. Creating a Memory-Optimized Table Type and Memory-Optimized Table Variable

```
create type dbo.mtvIDList as table
(
    ID int not null
        primary key nonclustered hash
        with (bucket_count=10000)
)
with (memory_optimized=on)
go

declare
    @IDList dbo.mtvIDList
```

You can pass memory-optimized table variables as table-valued parameters (TVP) to natively compiled and regular T-SQL procedures. As with on-disk based table-valued parameters, it is a very efficient way to pass a batch of rows to a T-SQL routine.

■ **Note** I will discuss the scenarios of passing a batch of rows to T-SQL routines and using memory-optimized table variables as the replacement of temporary tables in greater detail in Chapter 11.

You can use memory-optimized table variables to imitate row-by-row processing using cursors, which are not supported in natively compiled stored procedures. Listing 6-13 illustrates an example of using a memory-optimized table variable to imitate a static cursor. Obviously, it is better to avoid cursors and use set-based logic if at all possible.

Listing 6-13. Using a Memory-Optimized Table Variable to Imitate a Cursor

```
create type dbo.MODataStage as table
(
    ID int not null
        primary key nonclustered
        hash with (bucket_count=1000),
    Value int null
)
with (memory_optimized=on)
go

create proc dbo.CursorDemo
with native_compilation, schemabinding, execute as owner
as
begin atomic
with
```

```
(
    transaction isolation level = snapshot
    ,language=N'English'
)
    declare
        @tblCursor dbo.MODataStage
        ,@ID int = -1
        ,@Value int
        ,@RC int = 1

    /* Staging data in temporary table to imitate STATIC cursor */
    insert into @tblCursor(ID, Value)
        select ID, Value
        from dbo.MOData

    while @RC = 1
    begin
        select top 1 @ID = ID, @Value = Value
        from @tblCursor
        where ID > @ID
        order by ID

        select @RC = @@rowcount
        if @RC = 1
        begin
            /* Row processing */
            update dbo.MOData set Value = Value * 2 where ID = @ID
        end
    end
end
```

Summary

SQL Server uses native compilation to minimize the processing overhead of the interpreted T-SQL language. It generates separate DLLs for every memory-optimized object and loads it into process memory.

SQL Server supports native compilation of regular T-SQL stored procedures. It compiles them into DLLs at creation time or, in the case of a server or database restart, at the time of the first call. SQL Server optimizes natively compiled stored procedures and embeds an execution plan into the code. That plan never changes unless the procedure is recompiled after a SQL Server or database restart. You should drop and recreate procedures if data distribution has been significantly changed after compilation.

While natively compiled stored procedures are incredibly fast, they support a limited set of T-SQL language features. You can avoid such limitations by using interpreted T-SQL code that accesses memory-optimized tables through the query interop component of SQL Server. Almost all T-SQL language features are supported in this mode.

Memory-optimized table types and memory-optimized table variables are the in-memory analog of table types and table variables. They live in memory only, and they do not use `tempdb`. You can use memory-optimized table variables as a staging area for the data and to pass a batch of rows to a T-SQL routine. Memory-optimized table types allow you to create indexes similar to memory-optimized tables.

CHAPTER 7

Transaction Processing in In-Memory OLTP

This chapter discusses transaction processing in In-Memory OLTP. It elucidates what isolation levels are supported with native compilation and cross-container transactions, provides an overview of concurrency phenomena encountered in the database systems, and explains how In-Memory OLTP addresses them. Finally, this chapter talks about the lifetime of In-Memory OLTP transactions in detail.

ACID, Transaction Isolation Levels, and Concurrency Phenomena Overview

Transactions are the unit of work that read and modify data in a database and help to enforce consistency and durability of the data in a system. Every transaction in a properly implemented transaction management system has four different characteristics known as *atomicity*, *consistency*, *isolation*, and *durability*, often referenced as *ACID*.

- *Atomicity* guarantees that each transaction executes as an "all or nothing" approach. All changes done within a transaction are either committed or rolled back in full. Consider the classic example of transferring money between checking and savings bank accounts. That action consists of two separate operations: decreasing the balance of the checking account and increasing the balance of the savings account. Transaction atomicity guarantees that both operations either succeed or fail together, and a system will never be in the situation when money was deducted from the checking account but never added to the savings account.
- *Consistency* ensures that any database transaction brings the database from one consistent state to another and none of defined database rules and constraints were violated.

- *Isolation* ensures that the changes done in the transaction are isolated and invisible to other transactions until the transaction is committed. By the book, transaction isolation should guarantee that concurrent execution of the multiple transactions should bring the system to the same state as if those transactions were executed serially. However, in most database systems, such a requirement is often relaxed and controlled by *transaction isolation levels*.
- *Durability* guarantees that after a transaction is committed, all changes done by the transaction stay permanent and will survive a system crash. SQL Server achieves durability by using *Write-Ahead Logging* hardening log records in transaction log synchronously with data modifications.

The isolation requirements are the most complex to implement in multi-user environments. Even though it is possible to completely isolate different transactions from each other, this could lead to a high level of blocking and other concurrency issues in systems with volatile data. SQL Server addresses this situation by introducing several transaction isolation levels that relax isolation requirements at the cost of possible concurrency phenomena related to read data consistency:

- **Dirty Reads**: A transaction reads uncommitted (dirty) data from other uncommitted transactions.
- **Non-Repeatable Reads**: Subsequent attempts to read the same data from within the same transaction return different results. This data inconsistency issue arises when the other transactions modified, or even deleted, data between the reads done by the affected transaction.
- **Phantom Reads**: This phenomenon occurs when subsequent reads within the same transaction return new rows (the ones that the transaction did not read before). This happens when another transaction inserted the new data in between the reads done by the affected transaction.

Table 7-1 shows the data inconsistency issues that are possible for different transaction isolation levels. It is worth mentioning that every isolation level resolves write/write conflicts, preventing multiple active transactions from updating the same rows simultaneously.

Table 7-1. *Transaction Isolation Levels and Concurrency Phenomena*

Isolation Level	Dirty Reads	Non-Repeatable Reads	Phantom Reads
READ UNCOMMITTED	YES	YES	YES
READ COMMITTED	NO	YES	YES
REPEATABLE READ	NO	NO	YES
SERIALIZABLE	NO	NO	NO
SNAPSHOT	NO	NO	NO

With the exception of the SNAPSHOT isolation level, SQL Server uses locking to address concurrency phenomena when dealing with on-disk tables. When a transaction modifies a row, it acquires exclusive (X) locks on the row and holds it until the end of the transaction. That exclusive (X) lock prevents other sessions from accessing uncommitted data until the transaction is completed and the locks are released. This behavior is also known as *pessimistic concurrency*.

Such behavior also means that, in the case of a write/write conflict, the last modification wins. For example, when two transactions are trying to modify the same row, SQL Server blocks one of them until another transaction is committed, allowing blocked transactions to modify the data afterwards. No errors or exceptions are raised; however, changes done by the first transaction are overwritten.

In the case of on-disk tables and pessimistic concurrency, transaction isolation levels control how a session acquires and releases shared (S) locks when reading the data. Table 7-2 demonstrates that behavior.

Table 7-2. *Transaction Isolation Levels and Shared (S) Locks Behavior with On-disk Tables*

Isolation Level	Shared (S) Locks Behavior	Comments
READ UNCOMMITTED	(S) locks not acquired	Transaction can see uncommitted changes from the other sessions (dirty reads)
READ COMMITTED	(S) locks acquired and released immediately	Transaction will be blocked when it tries to read uncommitted rows with exclusive (X) locks held by the other sessions (no dirty reads)
REPEATABLE READ	(S) locks acquired and held till the end of transaction	Other sessions cannot modify a row after it was read (no non-repeatable reads). However, they can still insert phantom rows
SERIALIZABLE	Range (S) locks acquired and held till end of transaction	Other sessions cannot modify a row after it was read nor insert new rows in between rows that were read (no non-repeatable or phantom reads)

The SNAPSHOT isolation level uses a row-versioning model by creating the new version of the row after modification. In this model, all data modifications done by other transactions are invisible to the transaction after it starts.

Though it is implemented differently in the case of on-disk and memory-optimized tables, logically it behaves the same. A transaction will read a version of the row valid at the time when the transaction started, and sessions do not block each other. However, when two transactions try to update the same data, one of them will be aborted and rolled back to resolve the write/write conflict. This behavior is known as *optimistic concurrency*.

■ **Note** While SERIALIZABLE and SNAPSHOT isolation levels provide the same level of protection against data inconsistency issues, there is a subtle difference in their behavior. A SNAPSHOT isolation level transaction sees data as of the beginning of a transaction. With the SERIALIZABLE isolation level, the transaction sees data as of the time when the data was accessed for the first time.

Consider the situation when a session is reading data from a table in the middle of a transaction. If another session changed the data in that table after the transaction started but before data was read, the transaction in the SERIALIZABLE isolation level would see the changes while the SNAPSHOT transaction would not.

Transaction Isolation Levels in In-Memory OLTP

In-Memory OLTP supports three transaction isolation levels: SNAPSHOT, REPEATABLE READ, and SERIALIZABLE. However, In-Memory OLTP uses a completely different approach to enforce data consistency rules as compared to on-disk tables. Rather than block or being blocked by other sessions, In-Memory OLTP validates data consistency at the transaction COMMIT time and throws an exception and rolls back the transaction if rules were violated.

- In the SNAPSHOT isolation level, any changes done by other sessions are invisible to the transaction. A SNAPSHOT transaction always works with a snapshot of the data as of the time when transaction started. The only validation at the time of commit is checking for primary key violations, which is called *snapshot validation*.
- In the REPEATABLE READ isolation level, In-Memory OLTP validates that the rows that were read by the transaction have not been modified or deleted by the other transactions. A REPEATABLE READ transaction would not be able to commit if this was the case. That action is called *repeatable read validation*.
- In the SERIALIZABLE isolation level, SQL Server performs repeatable read validation and also checks for phantom rows that were possibly inserted by the other sessions. This process is called *serializable validation*.

Let's look at a few examples that demonstrate this behavior. As a first step, shown in Listing 7-1, let's create a memory-optimized table and insert a few rows there.

Listing 7-1. Data Consistency and Transaction Isolation Levels: Table Creation

```
create table dbo.HKData
(
    ID int not null,
    Col int not null,

    constraint PK_HKData
    primary key nonclustered hash(ID)
    with (bucket_count=64),
)
with (memory_optimized=on, durability=schema_only);

insert into dbo.HKData(ID, Col)
values(1,1),(2,2),(3,3),(4,4),(5,5);
```

Table 7-3 shows how concurrency works in the `REPEATABLE READ` transaction isolation level.

Table 7-3. *Concurrency in the REPEATABLE READ Transaction Isolation Level*

Session 1	Session 2	Results

`begin tran`
`  select ID, Col`
`  from dbo.HKData`
`  with (repeatableread)`		

	`update dbo.HKData`
	`set Col = -2`
	`where ID = 2`	

`  select ID, Col`
`  from dbo.HKData`
`  with (repeatableread)`
		Return old version of a row (Col = 2)

`commit`		Msg 41305, Level 16, State 0, Line 0 The current transaction failed to commit due to a repeatable read validation failure.

`begin tran`
`  select ID, Col`
`  from dbo.HKData`
`  with (repeatableread)`		

	`insert into dbo.HKData` `values(10,10)`	

`  select ID, Col`
`  from dbo.HKData`
`  with (repeatableread)`
		Does not return new row (10,10)

`commit`		Success

As you can see, with memory-optimized tables, other sessions were able to modify data that was read by the active REPEATABLE READ transaction. This led to a transaction abort at the time of COMMIT when the repeatable read validation failed. This is a completely different behavior than that of on-disk tables, where other sessions are blocked,unable to modify data until the REPEATABLE READ transaction successfully commits and releases shared (S) locks it held.

It is also worth noting that in the case of memory-optimized tables, the REPEATABLE READ isolation level protects you from the *Phantom Read* phenomenon, which is not the case with on-disk tables.

As a next step, let's repeat these tests in the SERIALIZABLE isolation level. You can see the code and the results of the execution in Table 7-4.

Table 7-4. *Concurrency in the SERIALIZABLE Transaction Isolation Level*

Session 1	Session 2	Results

`begin tran`
`  select ID, Col`
`  from dbo.HKData`
`  with (serializable)`		

	`update dbo.HKData`
	`set Col = -2`
	`where ID = 2`	

`  select ID, Col`
`  from dbo.HKData`
`  with (serializable)`
		Return old version of a row (Col = 2)

`commit`		Msg 41305, Level 16, State 0, Line 0 The current transaction failed to commit due to a repeatable read validation failure.

`begin tran`
`  select ID, Col`
`  from dbo.HKData`
`  with (serializable)`		

	`insert into dbo.HKData` `values(10,10)`	

`  select ID, Col`
`  from dbo.HKData`
`  with (serializable)`
		Does not return new row (10,10)

`commit`		Msg 41325, Level 16, State 0, Line 0 The current transaction failed to commit due to a serializable validation failure.

As you can see, the SERIALIZABLE isolation level prevents the session from committing a transaction when another session inserted a new row and violated the serializable validation. Like the REPEATABLE READ isolation level, this behavior is different from that of on-disk tables, where the SERIALIZABLE transaction successfully blocks other sessions until it is done.

Finally, let's repeat the tests in the SNAPSHOT isolation level. The code and results are shown in Table 7-5.

Table 7-5. *Concurrency in the SNAPSHOT Transaction Isolation Level*

Session 1	Session 2	Results

`begin tran`
`  select ID, Col`
`  from dbo.HKData`
`  with (snapshot)`		

	`update dbo.HKData`
	`set Col = -2`
	`where ID = 2`	

`  select ID, Col`
`  from dbo.HKData`
`  with (snapshot)`
		Return old version of a row (Col = 2)

`commit`		Success

`begin tran`
`  select ID, Col`
`  from dbo.HKData`
`  with (snapshot)`		

	`insert into dbo.HKData` `values(10,10)`	

`  select ID, Col`
`  from dbo.HKData`
`  with (snapshot)`
		Does not return new row (10,10)

`commit`		Success

The SNAPSHOT isolation level behaves in a similar manner to on-disk tables, and it protects from the *Non-Repeatable Reads* and *Phantom Reads* phenomena. As you can guess, it does not need to perform repeatable read and serializable validations at the commit stage and, therefore, it reduces the load on SQL Server. However, there is still snapshot validation, which checks for primary key violations and is done in any transaction isolation level.

Table 7-6 shows the code that leads to the primary key violation condition. In contrast to on-disk tables, the exception is raised on the commit stage rather than at the time of the second INSERT operation.

Table 7-6. *Snapshot Validation*

Session 1	Session 2	Results

`begin tran`
`  insert into dbo.HKData`
`  with (snapshot)`
`  (ID, Col)`
`  values(100,100)`		

	`begin tran`
	`  insert into dbo.HKData`
	`  with (snapshot)`
	`  (ID, Col)`
	`  values(100,100)`	

`commit`		Successfully commit the first session
	`commit`	Msg 41325, Level 16, State 1, Line 0 The current transaction failed to commit due to a serializable validation failure.

It is worth mentioning that the error number and message are the same with the serializable validation failure even though SQL Server validated the different rule.

Write/write conflicts work the same way regardless of the transaction isolation level in In-Memory OLTP. SQL Server does not allow a transaction to modify a row that has been modified by other uncommitted transactions. Table 7-7 illustrates this behavior. The code uses the SNAPSHOT isolation level; however, the behavior does not change with different isolation levels.

Table 7-7. *Write/Write Conflicts in In-Memory OLTP*

Session 1	Session 2	Results

`begin tran`
`  select ID, Col`
`  from dbo.HKData`
`  with (snapshot)`		

	`begin tran`
	`  update dbo.HKData`
	`  with (snapshot)`
	`  set Col = -3`
	`  where ID = 2`
	`commit`	

`  update dbo.HKData`
`  with (snapshot)`
`  set Col = -2`
`  where ID = 2`
		Msg 41302, Level 16, State 110, Line 1
		The current transaction attempted to update a record that has been updated since this transaction started. The transaction was aborted.
		Msg 3998, Level 16, State 1, Line 1
		Uncommittable transaction is detected at the end of the batch. The transaction is rolled back.
		The statement has been terminated.

`begin tran`
`  select ID, Col`
`  from dbo.HKData`
`  with (snapshot)`		

	`begin tran`
	`  update dbo.HKData`
	`  with (snapshot)`
	`  set Col = -3`
	`  where ID = 2`	

`  update dbo.HKData`
`  with (snapshot)`
`  set Col = -2`
`  where ID = 2`
		Msg 41302, Level 16, State 110, Line 1
		The current transaction attempted to update a record that has been updated since this transaction started. The transaction was aborted.
		Msg 3998, Level 16, State 1, Line 1
		Uncommittable transaction is detected at the end of the batch. The transaction is rolled back.
		The statement has been terminated.

	`commit`	Successful commit of Session 2 transaction

Cross-Container Transactions

Any access to memory-optimized tables from interpreted T-SQL is done through the Query Interop Engine and leads to *cross-container transactions*. You can use different transaction isolation levels for on-disk and memory-optimized tables. However, not all combinations are supported. Table 7-8 illustrates possible combinations for transaction isolation levels in cross-container transactions.

Table 7-8. *Isolation Levels for Cross-Container Transactions*

Isolation Levels for On-Disk Tables	Isolation Levels for Memory-Optimized Tables
`READ UNCOMMITTED`, `READ COMMITTED`,`READ COMMITTED SNAPSHOT`	`SNAPSHOT`, `REPEATABLE READ`, `SERIALIZABLE`
`REPEATABLE READ`, `SERIALIZABLE`	`SNAPSHOT` only
`SNAPSHOT`	Not supported

As you already know, internal implementations of `REPEATABLE READ` and `SERIALIZABLE` isolation levels are very different for on-disk and memory-optimized tables. Data consistency rules with on-disk tables rely on locking while In-Memory OLTP uses pre-commit validation. In cross-container transactions, SQL Server only supports `SNAPSHOT` isolation levels for memory-optimized tables when on-disk tables require `REPEATABLE READ` or `SERIALIZABLE` isolation.

Moreover, SQL Server does not allow access to memory-optimized tables when on-disk tables require `SNAPSHOT` isolation. Cross-container transactions, in a nutshell, consist of two internal transactions: one for on-disk and another one for memory-optimized tables. It is impossible to start both internal transactions at exactly the same time and guarantee the state of the data at the moment the transaction starts.

Listing 7-2 illustrates a transaction that tries to access data from memory-optimized and on-disk tables using incompatible transaction isolation levels.

Listing 7-2. Using Incompatible Isolation Levels in a Cross-Container Transaction

```
select sum(OrderTotal)
from
(
    select OrderTotal
    from dbo.Orders with (repeatableread) /* Memory-Optimized table */
    where CustomerId = @CustomerId

    union all

    select OrderTotal
    from dbo.OrderHistory with (repeatableread) /* On-Disk table */
    where CustomerId = @CustomerId
) o
```

As you already know, reading on-disk data in a REPEATABLE READ isolation level requires you to use SNAPSHOT isolation levels with memory-optimized tables and, therefore, the query from Listing 7-2 returns the error shown in Listing 7-3.

Listing 7-3. Incompatible Isolation Levels in a Cross-Container Transaction

```
Msg 41333, Level 16, State 1, Line 3

The following transactions must access memory optimized tables
and natively compiled stored procedures under snapshot isolation:
RepeatableRead transactions, Serializable transactions, and transactions
that access tables that are not memory optimized in RepeatableRead or
Serializable isolation.
```

As the general guideline, it is recommended to use the READ COMMITTED/SNAPSHOT combination in cross-container transactions during the regular workload. This combination provides the minimal blocking and least pre-commit overhead and should be acceptable in a large number of use cases. Other combinations are more appropriate during data migrations when it is important to avoid non-repeatable and phantom reads phenomena.

As you may have already noticed, SQL Server requires you to specify the transaction isolation level with a table hint when you are accessing memory-optimized tables. This does not apply to individual statements that execute outside of the explicitly started (with BEGIN TRAN) transaction. Those statements are called *autocommitted transactions,* and each of them executes in a separate transaction that is active for the duration of the statement execution. Listing 7-4 illustrates code with three statements. Each of them will run in their own autocommitted transactions.

Listing 7-4. Autocommitted Transactions

```
delete from dbo.HKData;

insert into dbo.HKData(ID, Col)
values(1,1),(2,2),(3,3),(4,4),(5,5);

select ID, Col
from dbo.HKData;
```

An isolation level hint is not required for statements running in autocommitted transactions. When the hint is omitted, the statement runs in the SNAPSHOT isolation level.

■ **Note** SQL Server allows you to keep a NOLOCK hint while accessing memory-optimized tables from autocommitted transactions. That hint is ignored. A READUNCOMMITTED hint, however, is not supported and triggers an error.

There is the useful database option `MEMORY_OPTIMIZED_ELEVATE_TO_SNAPSHOT`, which is disabled by default. When this option is enabled, SQL Server allows you to omit the isolation level hint in non-autocommitted transactions promoting them to the `SNAPSHOT` isolation level as with autocommitted transactions. Consider enabling this option when you migrate an existing system to In-Memory OLTP and have T-SQL code that accesses tables that become memory-optimized.

Transaction Lifetime

Although I have already discussed a few key elements used by In-Memory OLTP to manage data access and the concurrency model, let's review them here.

- The Global Transaction Timestamp is an auto-incremented value that uniquely identifies every transaction in the system. SQL Server increments and obtains this value at the transaction pre-commit stage.
- `TransactionId` is another identifier (timestamp) that also uniquely identifies a transaction. SQL Server obtains and increments its value at the moment when the transaction starts.
- Every row has `BeginTs` and `EndTs` timestamps, which correspond to the Global Transaction Timestamp of the transaction that inserted or deleted this version of a row.

Figure 7-1 shows the lifetime of a transaction that works with memory-optimized tables.

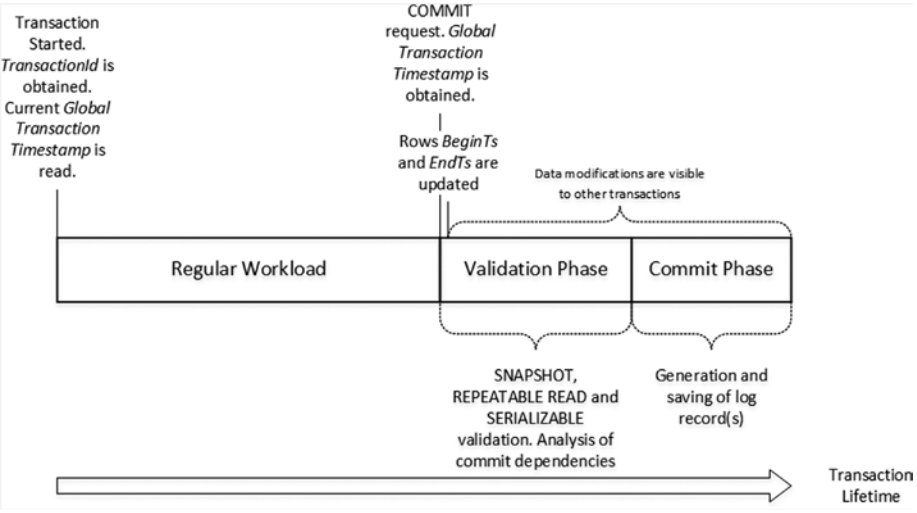


Figure 7-1. *Transaction lifetime*

At the time when a new transaction starts, it generates a new `TransactionId` and obtains the current Global Transaction Timestamp value. The Global Transaction Timestamp value dictates what rows are visible to the transaction, and it should be in between the `BeginTs` and `EndTs` for the rows to be visible. During data modifications, however, the transaction analyzes if there are any uncommitted versions of the rows, which prevents write/write conflicts when multiple sessions modify the same data.

When a transaction needs to delete a row, it updates the `EndTs` timestamp with the `TransactionId` value, which also has a flag that the timestamp contains the `TransactionId` rather than the Global Transaction Timestamp. The Insert operation creates a new row with the `BeginTs` of the`TransactionId` and the `EndTs` of `Infinity`. Finally, the update operation consists of delete and insert operations internally.

Figure 7-2 shows the data rows after we created and populated the `dbo.HKData` table in Listing 7-1, assuming that the rows were created by a transaction with the Global Transaction Timestamp of 5. (The hash index structure is omitted for simplicity's sake.)

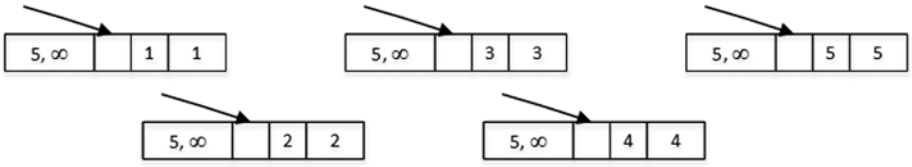


Figure 7-2. *Data in the dbo.HKData table after insert*

Let's assume that you have a transaction that started at the time when the Global Transaction Timestamp value was 10 and the `TransactionId` generated as -8. (I am using a negative value for `TransactionId` to illustrate the difference between two types of timestamps in the figures.)

Let's assume that the transaction performs the operations shown in Listing 7-5. The explicit transaction has already started, and the `BEGIN TRAN` statement is not included in the listing. All three statements are executing in the context of a single active transaction.

Listing 7-5. Data Modification Operations

```
insert into dbo.HKData with (snapshot)
(ID, Col)
values(10,10);

update dbo.HKData with (snapshot)
set Col = -2
where ID = 2;

delete from dbo.HKData with (snapshot)
where ID = 4;
```

Figure 7-3 illustrates the state of the data after data modifications. An `INSERT` statement created a new row, a `DELETE` statement updated the `EndTs` value in the row with `ID=4`, and an `UPDATE` statement changed the `EndTs` value of the row with `ID=2` and created a new version of the row with the same `ID`.

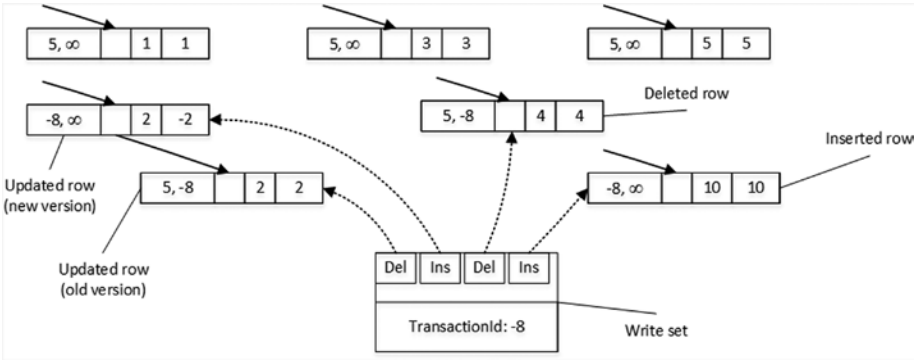


Figure 7-3. *Data in the dbo.HKData table after modifications*

It is important to note that the transaction maintains a *write set*, or pointers to rows that have been inserted and deleted by a transaction, which is used to generate transaction log records.

In addition to the write set, in the `REPEATABLE READ` and `SERIALIZABLE` isolation levels, transactions maintain a *read set* of the rows read by a transaction and use it for repeatable read validation. Finally, in the `SERIALIZABLE` isolation level, transactions maintain a *scan set*, which contains the information about predicates used by the queries in transaction. The scan set is used for serializable validation.

When a `COMMIT` request is issued, the transaction starts the validation phase. First, it generates a new Global Transaction Timestamp value and replaces the `TransactionId` with this value in all `BeginTs` and `EndTs` timestamps in the rows it modified. Figure 7-4 illustrates this action, assuming that the Global Transaction Timestamp value is 11.

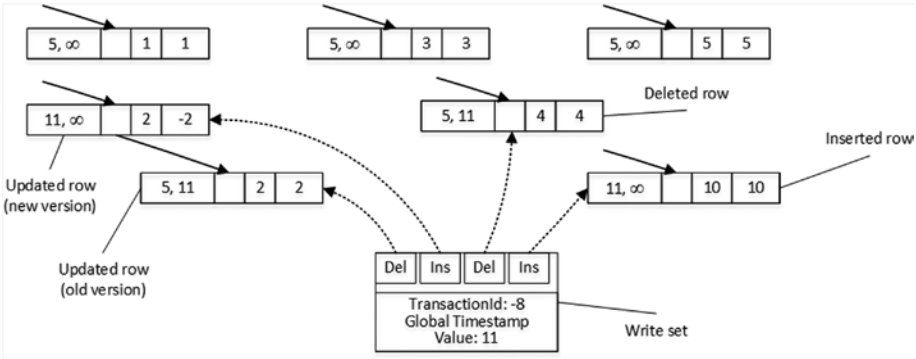


Figure 7-4. *Validation phase after BeginTs and EndTs values are replaced*

At this moment, the rows modified by transactions become visible to other transactions in the system even though the transaction has yet to be committed. Other transactions can see uncommitted rows, which leads to a situation called *commit dependency*. These transactions are not blocked at the time when they access those rows;

however, they do not return data to clients nor commit until the original transaction on which they have a commit dependency commits itself. I will talk about commit dependencies shortly.

As the next step, the transaction starts a validation phase. SQL Server performs several validations based on the isolation level of the transaction, as shown in Table 7-9.

Table 7-9. *Validations Done in the Different Transaction Isolation Levels*

	Snapshot Validation	Repeatable Read Validation	Serializable Validation
	Checking for primary key violations	Checking for non-repeatable reads	Checking for phantom reads
SNAPSHOT	YES	NO	NO
REPEATABLE READ	YES	YES	NO
SERIALIZABLE	YES	YES	YES

■ **Important** Repeatable read and serializable validations add an overhead to the system. Do not use `REPEATABLE READ` and `SERIALIZABLE` isolation levels unless you have a legitimate use case for such data consistency. We will discuss two of those use cases, such as supporting uniqueness and referential integrity, in Chapter 11.

After the required rules have been validated, the transaction waits for the commit dependencies to clear and the transaction on which it depends to commit. If those transactions fail to commit for any reason, such as a validation rules violation, the dependent transaction is also be rolled back and an error 41301 is generated.

Figure 7-5 illustrates a commit dependency scenario. Transaction `Tx2` can access uncommitted rows from transaction `Tx1` during `Tx1` validation and commit phases and, therefore, `Tx2` has commit dependency on `Tx1`. After the `Tx2` validation phase is completed, `Tx2` has to wait for `Tx1` to commit and the commit dependency to clear before entering the commit phase.

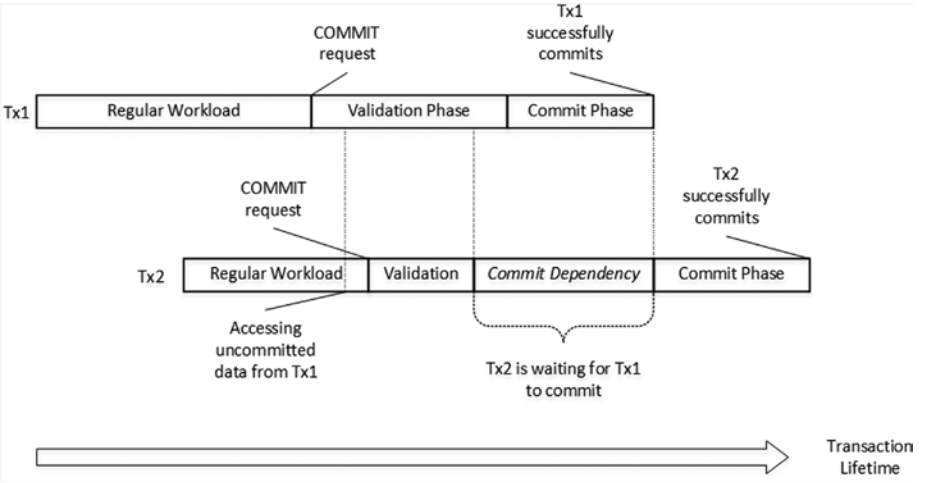


Figure 7-5. *Commit Dependency: Successful Commit*

If Tx1, for example, failed to commit due to serializable validation violation, Tx2 would be rolled back with Error 41301, as shown in Figure 7-6.

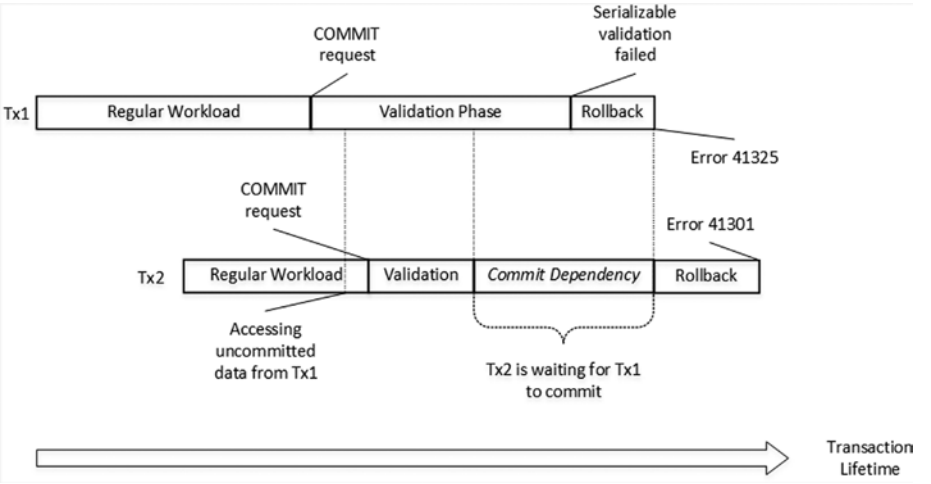


Figure 7-6. *Commit Dependency: Validation Error*

■ **Note** You can track commit dependencies using the `dependency_acquiredtx_event` and `waiting_for_dependenciestx_event` extended events.

Finally, when all commit dependencies are cleared, the transaction moves to the commit phase, generates one or more log records, saves them to the transaction log, and completes the transaction.

Commit dependency is technically the case of blocking in In-Memory OLTP. However, the validation and commit phases of the transactions are relatively short, and that blocking should not be excessive. It is also worth noting that transaction logging in In-Memory OLTP is more efficient compared to on-disk transactions. I will discuss it in more detail in Chapter 8.

■ **Note** You can read more about the concurrency model in In-Memory OLTP at https://msdn.microsoft.com/en-us/library/dn479429.aspx.

Summary

In-Memory OLTP supports three transaction isolation levels, `SNAPSHOT`, `REPEATABLE READ`, and `SERIALIZABLE`. In contrast to on-disk tables, where non-repeatable and phantom reads are addressed by acquiring and holding the locks, In-Memory OLTP validates data consistency rules on the transaction commit stage. An exception will be raised and the transaction will be rolled back if rules were violated.

Repeatable read and serializable validation adds an overhead to transaction processing. It is recommended to use the `SNAPSHOT` isolation level during a regular workload unless `REPEATABLE READ` or `SERIALIZABLE` data consistency is required.

You can use different transaction isolation levels for on-disk and memory-optimized tables in cross-container transactions; however, not all combinations are supported. The recommended practice is using the `READ COMMITTED` isolation level for on-disk and the `SNAPSHOT` isolation level for memory-optimized tables.

SQL Server does not require you to specify transaction isolation level when you access memory-optimized tables through the interop engine in autocommitted (single statement) transactions. SQL Server automatically promotes such transactions to the `SNAPSHOT` isolation level. However, you should specify an isolation level hint when a transaction is explicitly started with `BEGIN TRAN` statement. You can avoid this by enabling the `MEMORY_OPTIMIZED_ELEVATE_TO_SNAPSHOT` database option. This option is useful when you migrate existing system to use In-Memory OLTP.

CHAPTER 8

Data Storage, Logging, and Recovery

This chapter discusses how In-Memory OLTP stores the data from durable memory-optimized tables on disk. It illustrates the concept of checkpoint file pairs used by SQL Server to persist the data, provides an overview of checkpoint process in In-Memory OLTP, and discusses recovery of memory-optimized data. Finally, this chapter demonstrates how In-Memory OLTP logs the data in a transaction log and why In-Memory OLTP logging is more efficient compared to on-disk tables.

Data Storage

The data from durable memory-optimized tables is stored separately from on-disk tables. SQL Server uses a streaming mechanism to store it, which is based on `FILESTREAM` technology. In-Memory OLTP and `FILESTREAM`, however, store data separately from each other and you should have two separate filegroups: one for In-Memory OLTP and another for `FILESTREAM` data when the database uses both technologies.

There is a conceptual difference between how on-disk and memory-optimized data are stored. On-disk tables store the single, most recent version of the row. Multiple updates of the row change the same row object multiple times. Deletion of the row removes it from the database. Finally, it is always possible to locate a data row in a data file when needed.

In-Memory OLTP uses a completely different approach and persists multiple versions of the row on disk. Multiple updates of the data row generate multiple row objects, each of which has a different lifetime. SQL Server appends them to binary files stored in the In-Memory OLTP filegroup, which are called *checkpoint file pairs* (CFP).

It is impossible to predict where a data row is stored in checkpoint file pairs. Nor are there use cases for such an operation. The only purposes these files serve is to provide data durability and improve the performance of loading data into memory on database startup.

As you can guess by the name, each checkpoint file pair consists of two files: a *data file* and a *delta file*. Each CFP covers operations for a range of Global Transaction Timestamp values, logging operations on the rows that have `BeginTs` in this range. Every time you insert a row, it is saved into a data file. Every time you delete a row,

the information about the deleted row is saved into a delta file. An update generates two operations, `INSERT` and `DELETE`, and it saves this information to both files. Figure 8-1 provides a high-level overview of the structure of checkpoint file pairs.

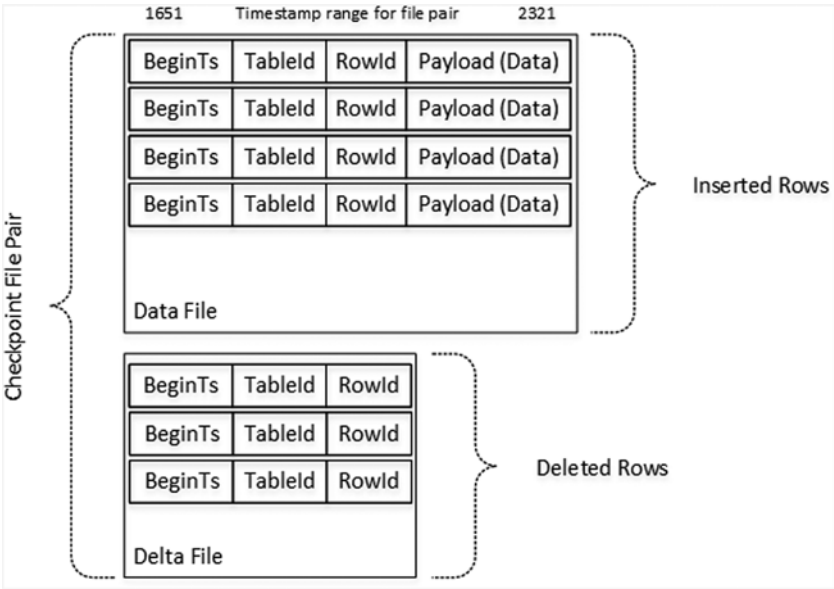


Figure 8-1. *Data in checkpoint files*

Figure 8-2 shows an example of a database with six check point file pairs in the different states. The vertical rectangles with a solid fill represent data files. The rectangles with a dotted fill represent delta files. This is just an illustration. In reality, every database will have at least eight checkpoint file pairs in the various states, which we will cover in detail shortly.

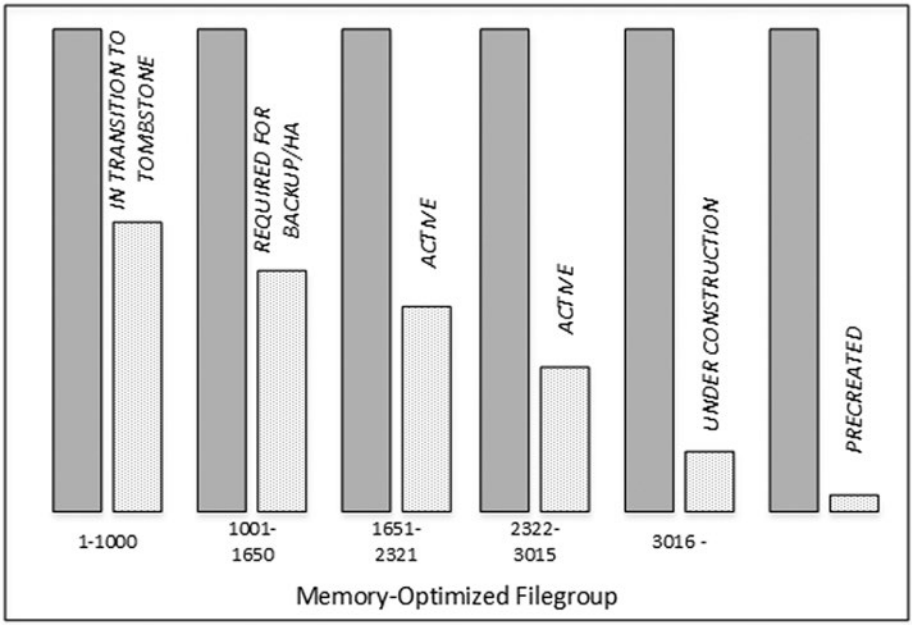


Figure 8-2. *A database with multiple checkpoint file pairs*

Internally, SQL Server stores checkpoint file pair metadata in an 8,192 slot array. Even though, in theory, it allows you to store up to 8,192 * 128MB = 1TB of data, Microsoft does not recommend nor support configurations with more than 256GB of data stored in durable memory-optimized tables. There is no restriction on the amount of data stored in non-durable memory-optimized tables.

Using a separate delta file to log deletions allows SQL Server to avoid modifications in data files and random I/O in cases when rows are deleted. Both data and delta files are append-only. Moreover, when files are closed (again, more on this shortly), they become read-only.

Checkpoint File Pairs States

Each checkpoint file pair can be in one of several states during its lifetime, as illustrated in Figure 8-3.

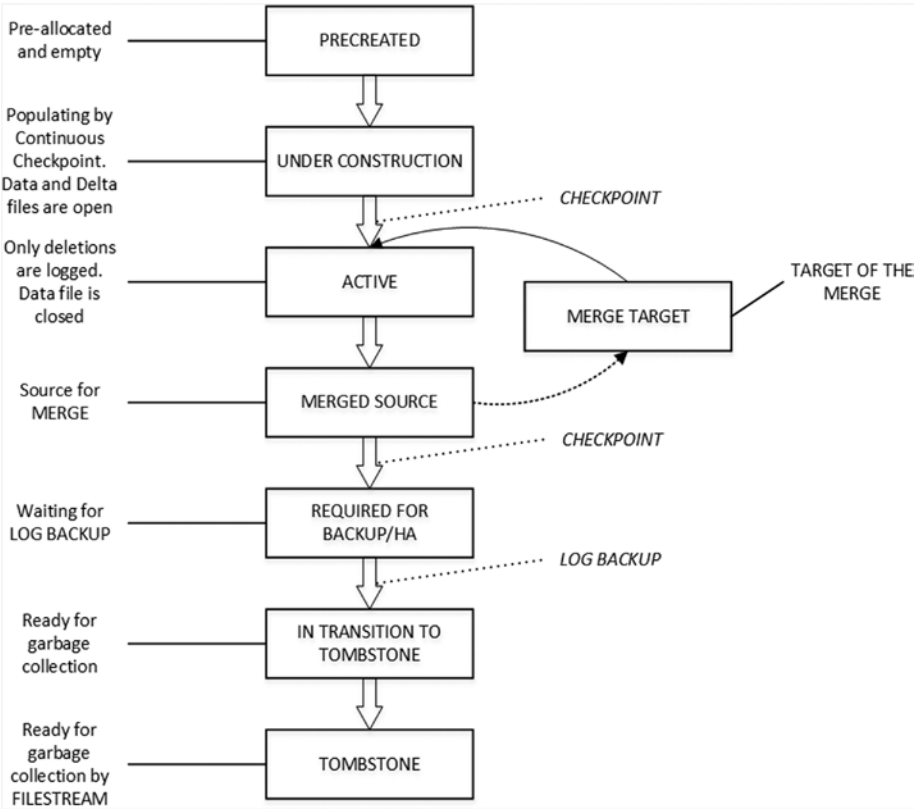


Figure 8-3. *Checkpoint file pair states*

Let's look at all of these states in more detail.

PRECREATED CFP State

When you create the first memory-optimized table in the database, SQL Server generates the set of checkpoint file pairs. Those files are empty and they are created to minimize wait time when new files are needed.

The total number of new files is based on the storage and hardware configuration. SQL Server creates a separate checkpoint file pair per scheduler (logical CPU) with a minimum of eight CFPs. The initial size of the files is based on the amount of server memory, as shown in Table 8-1.

Table 8-1. *Initial Size of Checkpoint Files*

Server Memory	Data File Size	Delta File Size
Less than 16GB	16MB	1MB
16GB or more	128MB	8MB

UNDER CONSTRUCTION CFP State and CHECKPOINT Process

As you already know, SQL Server uses the transaction log to persist information about data modifications in the database. Transaction log records can be used to reconstruct any data changes in the event of an unexpected shutdown or crash; however, that process can be very time-consuming if a large number of log records need to be replayed.

SQL Server uses *checkpoints* to mitigate that problem. Even though on-disk and In-Memory OLTP checkpoint processes are independent from each other, they do the same thing: persist the data changes on disk, reducing database recovery time. The last checkpoint identifies up to which point the data changes have been persisted and from which log records need to be replayed.

With on-disk tables, the frequency of checkpoint operations depends on the server-level `recovery interval` and database-level `TARGET_RECOVERY_TIME` settings. While such an approach helps SQL Server to improve write performance by batching multiple random I/O writes together, it leads to spikes in I/O activity at the time when checkpoint occurs.

In contrast, In-Memory OLTP implements *continuous checkpoints*. It continuously scans the transaction log, streaming and appending the changes to checkpoint file pairs in the `UNDER CONSTRUCTION` state. The new versions of the rows are appended to the data files and deletions are appended to delta files. The continuous checkpoint also appends information about deletions to CFPs in the `ACTIVE` state, which we will discuss shortly.

As mentioned, the rows in checkpoint file pairs are never updated. Instead, the new row version is appended to the data file and the old version is marked as *deleted* in the delta files. This leads to sequential streaming I/O, which is significantly faster compared to random I/O even in the case of SSD drives.

ACTIVE CFP State

The continuous checkpoint process *continuously* persists the data from memory-optimized tables on disk. However, there is still a *checkpoint event*, which performs several actions.

- It scans the remaining (unscanned) portion of the transaction log and hardens all remaining log records to checkpoint file pairs. You can consider it as the continuous checkpoint catching up and processing the remaining part of the log up to the checkpoint event.
- It transitions all `UNDER CONSTRACTION` CFPs to the `ACTIVE` state.

- It creates a *checkpoint inventory*, which contains the information about all files from the previous checkpoint along with any files added by the current checkpoint. The checkpoint inventory and the Global Transaction Timestamp are hardened in the transaction log and available to SQL Server during the recovery process. The combination of ACTIVE CFPs and the tail of the log allow SQL Server to recover the data from memory-optimized tables if needed.

The checkpoint event changes the state of all `UNDER CONSTRUCTION` checkpoint file pairs to `ACTIVE`. SQL Server does not stream new data into `ACTIVE` CFPs so data files become read-only; however, it still uses the delta files storing the information about the `DELETE` and `UPDATE` operations that occurred against the versions of the rows from the corresponding data files.

In the case of memory-optimized tables, a checkpoint is invoked either manually with the `CHECKPOINT` command, or automatically when the transaction log has grown more than 512MB since the last checkpoint. It is worth mentioning that SQL Server does not differentiate log activity between on-disk and memory-optimized tables when using this 512MB threshold. It is entirely possible that a checkpoint is triggered even when there were no transactions against memory-optimized tables.

Typically, the combined size of the `ACTIVE` checkpoint file pairs on disk is about twice the size of the durable memory-optimized tables in memory. However, in some cases, SQL Server may require more space to store memory-optimized data.

MERGE TARGET and MERGED SOURCE CFP States and Merge Process

Overtime, as data modifications progress, the percent of deleted rows in the `ACTIVE` checkpoint file pairs increases. This condition adds unnecessary storage overhead and slows down the data loading process during recovery. SQL Server addresses this situation with a process called *merge*.

A background task called the *Merge Policy Evaluator* periodically analyzes if adjacent `ACTIVE` CFPs can be merged together in a way that active, non-deleted rows from the merged data files would fit into the new 16MB or 128MB data file. When it happens, SQL Server creates the new CFP in a `MERGE TARGET` state and populates it with the data from the multiple `ACTIVE` CFPs, filtering out deleted rows.

Even though the Merge Policy Evaluator can identify multiple possible merges, every CFP can participate only in one of them. Table 8-2 shows several examples of the possible merges.

Table 8-2. *Merge Examples*

Adjacent Source Files (% Full)	Merge Results
CFP0(40%), CFP1(45%), CFP2(60%)	CFP0 + CFP1 (85%)
CFP0(10%), CFP1(15%), CFP2(70%), CFP3(10%)	CFP0 + CFP1 + CFP2 (95%)
CFP0(55%), CFP1(50%)	No Merge is done

In most cases, you can rely on the automatic merge process. However, you can trigger a manual merge using the `sys.sp_xtp_merge_checkpoint_files` stored procedure. You will see such an example in Appendix C.

Once the merge process is complete, the next checkpoint event transitions the `MERGE TARGET` CFP to `ACTIVE` and former `ACTIVE` CFPs to `MERGED SOURCE` states.

REQUIRED FOR BACKUP/HA, IN TRANSITION TO TOMBSTONE, and TOMBSTONE CFP States

After the next checkpoint event occurs, the `MERGED SOURCE` CFPs are no longer needed for database recovery. Former `MERGE TARGET` and now `ACTIVE` CFPs can be used for this purpose. However, those CFPs are still needed if you want to restore the database from a backup, so they are switched to the `REQUIRED FOR BACKUP/HA` state.

The checkpoint file pairs stay in that state until the log truncation point passed their LSNs. In `FULL` recovery model that means that a log backup has been taken, log records were sent to secondary nodes, and other processes that read transaction log have not fallen behind. Obviously, in a `SIMPLE` recovery model, log backup is not required and the log truncation point is controlled by checkpoints.

Once it happens, CFPs are transitioned to the `IN TRANSITION TO TOMBSTONE` state, where they become eligible for garbage collection. Another In-Memory OLTP background thread switches them to the `TOMBSTONE` state, which they stay in until they are deallocated by the `FILESTREAM` garbage collector thread.

■ **Note** In reality, it is possible that multiple log backups are required for the CFP to switch to the `IN TRANSITION TO TOMBSTONE` state.

As with the merge process, in most cases you can rely on automatic garbage collection in both In-Memory OLTP and FILESTREAM; however, you can force garbage collection using the `sys.sp_xtp_checkpoint_force_garbage_collection` and `sys.sp_filestream_force_garbage_collection` stored procedures. You can see these procedures in action in Appendix C.

■ **Note** You can analyze the state of a checkpoint file pair using the `sys.dm_db_xtp_checkpoint_files` DMV. Appendix C talks about this view in greater depth and shows how CFP states change through their lifetime.

Transaction Logging

As mentioned in the previous chapter, transaction logging in In-Memory OLTP is more efficient compared to Storage Engine. Both engines share the same transaction log and perform *write-ahead logging* (WAL); however, the log records format and algorithms are very different.

With on-disk tables, SQL Server generates transaction log records on a per-index basis. For example, when you insert a single row into a table with clustered and nonclustered indexes, it will log insert operations in every individual index separately. Moreover, it will log internal operations, such as extent and page allocations, page splits, and a few others.

All log records are saved in a transaction log and hardened on disk pretty much synchronously at the time when they were created. Even though every database has a cache called *Log Buffer* to batch log writes, that cache is very small, about 60KB. Moreover, some operations, such as `COMMIT` and `CHECKPOINT`, flush that cache whether it is full or not.

Finally, SQL Server has to include before-update (`UNDO`) and after-update (`REDO`) versions of the row to the log records. Checkpoint process is asynchronous and it does not check the state of transaction that modified the page. It is entirely possible for the checkpoint to save the dirty data pages from uncommitted transactions and the `UNDO` part of the log records are required to roll back the changes.

Transaction logging in In-Memory OLTP addresses these inefficiencies. The first major difference is that In-Memory OLTP generates and saves log records at the time of the transaction `COMMIT` rather than during each data row modification. Therefore, rolled-back transactions do not generate any log activity.

The format of a log record is also different. Log records do not include any `UNDO` information. Dirty data from uncommitted transactions will never materialize on disk and, therefore, In-Memory OLTP log data does not need to support the `UNDO` stage of crash recovery nor log uncommitted changes.

In-Memory OLTP generates log records based on the transactions write set. All data modifications are combined together in one or very few log records based on the write set and inserted rows' size.

Let's examine this behavior and run the code shown in Listing 8-1. It starts a transaction and inserts 500 rows into a memory-optimized table. Then it examines the content of the transaction log using the undocumented `sys.fn_dblog` system function.

Listing 8-1. Transaction Logging in In-Memory OLTP: Memory-Optimized Table Logging

```
create table dbo.HKData
(
    ID int not null,
    Col int not null,

    constraint PK_HKData
    primary key nonclustered hash(ID)
    with (bucket_count=1024),
)
with (memory_optimized=on, durability=schema_and_data)
go

declare
    @I int = 1

begin tran
    while @I <= 500
    begin
        insert into dbo.HKData with (snapshot)
           (ID, Col)
        values(@I, @I)

           set @I += 1
    end
commit
go

select *
from sys.fn_dblog(NULL, NULL)
order by [Current LSN];
```

Figure 8-4 illustrates the content of the transaction log. You can see the single transaction record for the In-Memory OLTP transaction.

	Current LSN	Operation	Context	Transaction ID	LogBlockGeneration	Tag Bits	Log Record Fixed Length	Log Record Length	Previou
1	0000001f:0000593b:0001	LOP_BEGIN_XACT	LCX_NULL	0000:000003eb	0	0x0000	76	144	000000
2	0000001f:0000593b:0002	LOP_HK	LCX_NULL	0000:000003eb	0	0x0000	28	9568	000000
3	0000001f:0000593b:0003	LOP_COMMIT_XACT	LCX_NULL	0000:000003eb	0	0x0000	80	84	000000

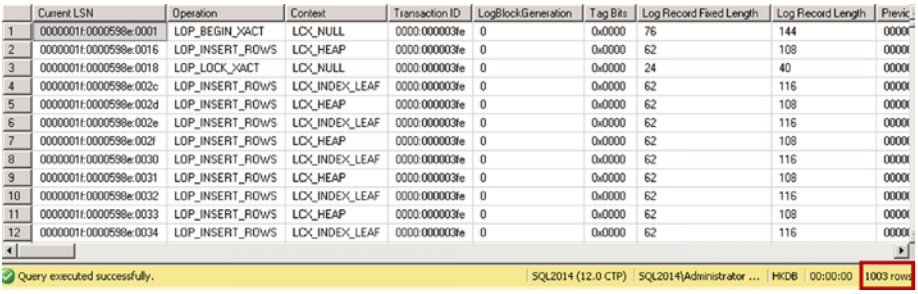
Figure 8-4. *Transaction log content after the In-Memory OLTP transaction*

Let's repeat this test with an on-disk table of a similar structure. Listing 8-2 shows the code that creates a table and populates it with data.

Listing 8-2. Transaction Logging in In-Memory OLTP: On-Disk Table Logging

```
create table dbo.DiskData
(
    ID int not null,
    Col int not null,

    constraint PK_DiskData
    primary key nonclustered(ID)
)
go

declare
    @I int = 1

begin tran
    while @I <= 500
    begin
        insert into dbo.DiskData(ID, Col)
        values(@I, @I)

        set @I += 1
    end
commit
```

As you can see in Figure 8-5, the same transaction generated more than 1,000 log records.



	Current LSN	Operation	Context	Transaction ID	LogBlockGeneration	Tag Bits	Log Record Fixed Length	Log Record Length	Previc
1	0000001f:0000598e:0001	LOP_BEGIN_XACT	LCX_NULL	0000:000003fe	0	0x0000	76	144	0000(
2	0000001f:0000598e:0016	LOP_INSERT_ROWS	LCX_HEAP	0000:000003fe	0	0x0000	62	108	0000(
3	0000001f:0000598e:0018	LOP_LOCK_XACT	LCX_NULL	0000:000003fe	0	0x0000	24	40	0000(
4	0000001f:0000598e:002c	LOP_INSERT_ROWS	LCX_INDEX_LEAF	0000:000003fe	0	0x0000	62	116	0000(
5	0000001f:0000598e:002d	LOP_INSERT_ROWS	LCX_HEAP	0000:000003fe	0	0x0000	62	108	0000(
6	0000001f:0000598e:002e	LOP_INSERT_ROWS	LCX_INDEX_LEAF	0000:000003fe	0	0x0000	62	116	0000(
7	0000001f:0000598e:002f	LOP_INSERT_ROWS	LCX_HEAP	0000:000003fe	0	0x0000	62	108	0000(
8	0000001f:0000598e:0030	LOP_INSERT_ROWS	LCX_INDEX_LEAF	0000:000003fe	0	0x0000	62	116	0000(
9	0000001f:0000598e:0031	LOP_INSERT_ROWS	LCX_HEAP	0000:000003fe	0	0x0000	62	108	0000(
10	0000001f:0000598e:0032	LOP_INSERT_ROWS	LCX_INDEX_LEAF	0000:000003fe	0	0x0000	62	116	0000(
11	0000001f:0000598e:0033	LOP_INSERT_ROWS	LCX_HEAP	0000:000003fe	0	0x0000	62	108	0000(
12	0000001f:0000598e:0034	LOP_INSERT_ROWS	LCX_INDEX_LEAF	0000:000003fe	0	0x0000	62	116	0000(

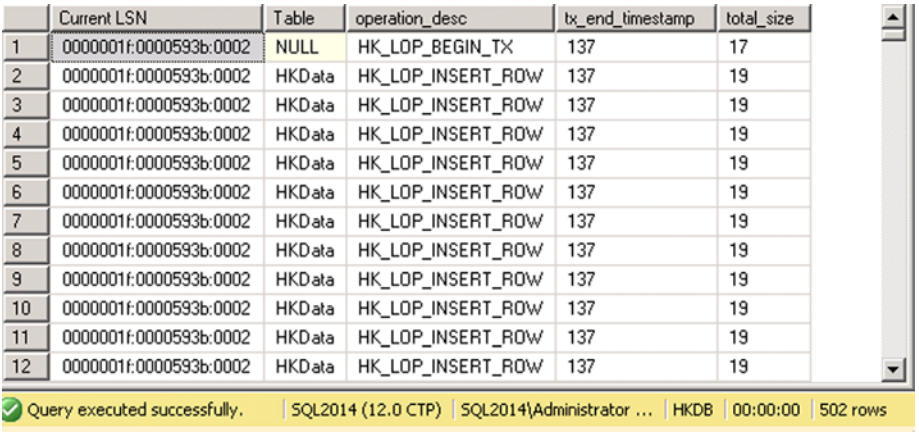
Query executed successfully. | SQL2014 (12.0 CTP) | SQL2014\Administrator ... | HKDB | 00:00:00 | 1003 rows

Figure 8-5. *Transaction log content after on-disk table modification*

You can use another undocumented function, `sys.fn_dblog_xtp`, to examine the logical content of an In-Memory OLTP log record. Listing 8-3 shows the code that utilizes this function and Figure 8-6 shows the output of that code. You should use the LSN of the `LSN_HK` log record from the Listing 8-2 output as the parameter of the function.

Listing 8-3. Analyzing an In-Memory OLTP Log Record

```
select [Current LSN], object_name(table_id) as [Table]
        ,operation_desc, tx_end_timestamp, total_size
from sys.fn_dblog_xtp
(
    '0x0000001f:0000593b:0002'
    ,'0x0000001f:0000593b:0002'
)
```



	Current LSN	Table	operation_desc	tx_end_timestamp	total_size
1	0000001f:0000593b:0002	NULL	HK_LOP_BEGIN_TX	137	17
2	0000001f:0000593b:0002	HKData	HK_LOP_INSERT_ROW	137	19
3	0000001f:0000593b:0002	HKData	HK_LOP_INSERT_ROW	137	19
4	0000001f:0000593b:0002	HKData	HK_LOP_INSERT_ROW	137	19
5	0000001f:0000593b:0002	HKData	HK_LOP_INSERT_ROW	137	19
6	0000001f:0000593b:0002	HKData	HK_LOP_INSERT_ROW	137	19
7	0000001f:0000593b:0002	HKData	HK_LOP_INSERT_ROW	137	19
8	0000001f:0000593b:0002	HKData	HK_LOP_INSERT_ROW	137	19
9	0000001f:0000593b:0002	HKData	HK_LOP_INSERT_ROW	137	19
10	0000001f:0000593b:0002	HKData	HK_LOP_INSERT_ROW	137	19
11	0000001f:0000593b:0002	HKData	HK_LOP_INSERT_ROW	137	19
12	0000001f:0000593b:0002	HKData	HK_LOP_INSERT_ROW	137	19

Query executed successfully. | SQL2014 (12.0 CTP) | SQL2014\Administrator ... | HKDB | 00:00:00 | 502 rows

Figure 8-6. *In-Memory OLTP transaction log record details*

Finally, it is worth stating again that any data modification on non-durable tables (`DURABILITY=SCHEMA_ONLY`) is not logged in the transaction log nor is its data persisted on disk.

Recovery

During the recovery stage, SQL Server locates the most recent checkpoint inventory and passes it to the In-Memory OLTP Engine, which starts recovering memory-optimized data in parallel with on-disk tables. The In-Memory OLTP Engine obtains the list of all `ACTIVE` checkpoint file pairs and starts loading data from them. It loads only the non-deleted versions of rows using delta files as the filter. It checks that a row from a data file is not deleted and is not referenced in the delta files. Based on the results of this check, a row is either loaded to memory or discarded.

The process of loading data is highly scalable. SQL Server creates one thread per logical CPU, and each thread processes an individual checkpoint file pair. In a large number of cases, the performance of the I/O subsystem becomes the limiting factor in data-loading performance.

As the opposite of on-disk tables, indexes on memory-optimized tables are not persisted. As you remember, indexes in In-Memory OLTP are just the memory pointers, and the memory addresses of the rows change after they are reloaded into the memory. Therefore, indexes must be recreated during the recovery stage.

Figure 8-7 illustrates the data-loading process.

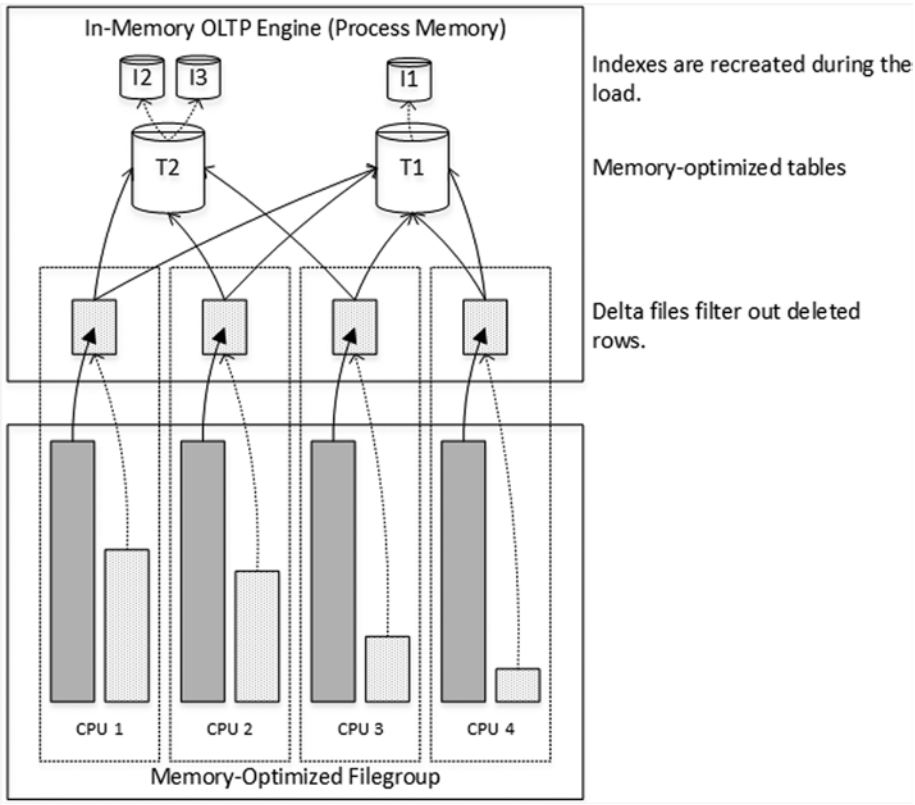


Figure 8-7. *Loading data to memory*

After the data from CFPs has been loaded, SQL Server completes the recovery by applying the changes from the tail of the transaction log, bringing the database back to the state as of the time of crash or shut down. As you already know, In-Memory OLTP does not log uncommitted changes and, therefore, no UNDO stage is required during the recovery.

Summary

The data from durable memory-optimized tables is placed into a separate file group utilizing `FILESTREAM` technology under the hood. The data is stored in the set of checkpoint file pairs. Each pair consists of two files, data and delta. Data files store the row version data. Delta files store the information about deleted rows.

The data in checkpoint file pairs is never updated. A `DELETE` operation generates the new entry in delta files. An `UPDATE` operation stores the new version of the row in the data file, marking the old version as deleted in the delta file. SQL Server utilizes the sequential streaming API to write data to those files without any random I/O involved.

Every checkpoint file pair covers a particular interval of Global Transaction Timestamps and goes through a set of predefines states. SQL Server stores the new row data in CFPs in the `UNDER CONSTRUCTION` state. These CFPs are converted to the `ACTIVE` state at a checkpoint event. Data files of `ACTIVE` CFPs are closed and they do not accept the new row versions; however, they still log the information about deletions in the delta files.

SQL Server merges the data from the `ACTIVE` checkpoint file pairs, filtering out deleted rows. After the merge is completed and the source CFPs are backed up, SQL Server marks them for garbage collection and deallocates them.

`ACTIVE` checkpoint file pairs are used during database recovery along with the tail of the log. The In-Memory OLTP recovery process is highly scalable and very fast. Indexes on memory-optimized tables are not persisted on disk and recreated when data is loaded into the memory.

Transaction logging in In-Memory OLTP is more efficient compared to on-disk tables. Transactions are logged at time of `COMMIT` based on the transaction write set. Log records are compact and contain information about multiple row-related operations.

CHAPTER 9

Garbage Collection

This chapter covers the garbage collection process used in the In-Memory OLTP Engine. It provides an overview of the various components involved in garbage collection and demonstrates how they interact with each other.

Garbage Collection Process Overview

In-memory OLTP is a row-versioning system. `UPDATE` operations generate new versions of rows rather than updating row data. `DELETE` operations do not remove the rows but rather update the `EndTs` row timestamp. Rows created by aborted transactions are not deallocated immediately and they stay as part of the index row chains even after rollback.

As you know, every row has two timestamps (`BeginTs` and `EndTs`) that indicate row lifetime: when the row was created and when it was deleted. Transactions can only see the versions of rows that were valid at the time when the transaction started. In practice, this means that a row is visible for the transaction only if the *Global Transaction Timestamp* value at the start of transaction is between the `BeginTs` and `EndTs` timestamps of the row.

At some point, when the `EndTs` timestamp of a row is older than the *Global Transaction Timestamp* of the *Oldest Active Transaction* in the system, the row *expires*. Expired rows are invisible for active transactions and eventually they need to be deallocated to reclaim system memory and speed up index chain navigation. This process is called *garbage collection*.

The garbage collection process in In-Memory OLTP has been designed with the following goals:

- **Non-blocking**: The garbage collection process should not block user threads and should produce minimal performance impact on the system.
- **Responsive**: The garbage collection process should react to memory pressure.
- **Cooperative and Scalable**: The garbage collection process should not rely on a single system thread to perform memory deallocation and should use regular worker threads during the process.

The cooperative nature of garbage collection makes it quite different from the *typical* SQL Server background processes. Even though there is a dedicated system garbage collection thread called the *idle worker thread*, the major part of the work is done by the regular user worker threads. This allows the process to scale and keep up with the workload in the system.

User threads participate in the garbage collection process in two different ways. They unlink old, expired rows from the row chains and perform actual deallocation. These actions are separate from each other, as you will see shortly.

Let's look at the process in detail. Figure 9-1 illustrates the logical structure of a table with two hash indexes on the `Name` and `City` columns. You saw this figure in previous chapters; however, in this chapter I added another element called `idxLinkCount`, which indicates in how many index chains the rows are participating. It is displayed with the underline font in the figure; note that all rows have a value of two, which corresponds to the number of indexes in the table.

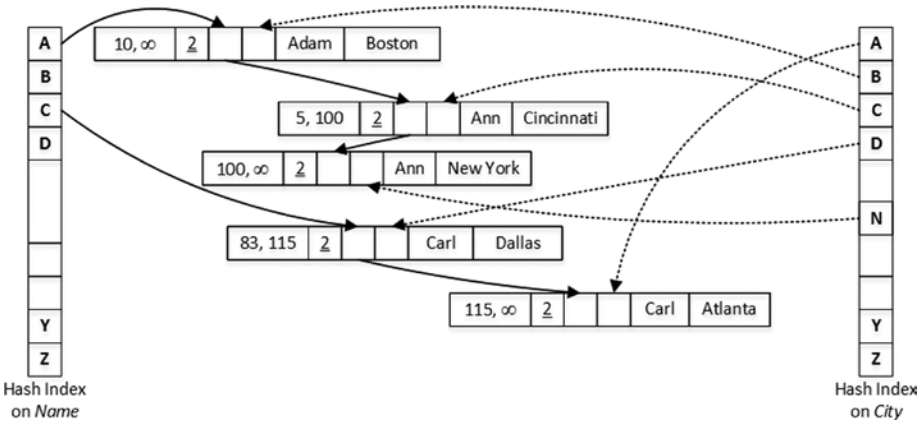


Figure 9-1. *Initial state of the data*

Assume that you have a session that runs two queries, as shown in Listing 9-1, at time when the *Oldest Active Transaction Timestamp* is 110 and the *Global Transaction Timestamp* is 125.

Listing 9-1. First Batch

```
select * from dbo.People where Name = 'Adam';
select * from dbo.People where Name = 'Carl';
```

The first `SELECT` scanned the `Name` index row chain for the bucket with value `A` and detected the `Ann` row with an `EndTs` of 100. The *Oldest Active Transaction Timestamp* is 110, so this row is expired and invisible for the active transactions in the system. As result, the user thread unlinked the row from the `Name` index row chain and decreased the `idxLinkCnt` value.

The second SELECT detects the deleted Carl row. However, the EndTs of this row is greater than the *Oldest Active Transaction Timestamp, so* this row can still be visible for some of the active transactions. Therefore, this row cannot be unlinked from the index chain. Figure 9-2 illustrates the state of the data after the execution of the queries.

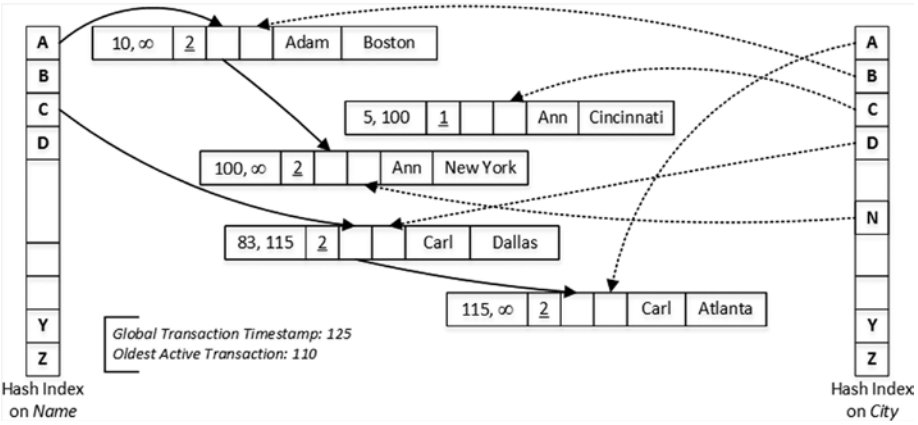


Figure 9-2. *State of the data after the first two queries*

Now, let's assume that some of the active transactions were completed and you ran the second batch of the queries from Listing 9-2 at the time when the *Oldest Active Transaction Timestamp* was 120 and the *Global Transaction Timestamp* was 130.

Listing 9-2. Second Batch

```
select * from dbo.People where City = 'Cincinatti';
select * from dbo.People where City = 'Dallas';
```

The first SELECT found the expired Ann row in the City index chain and removed it from there. At this point, the row is not participating in any row chains and, therefore, can be deallocated. However, the row is not deallocated immediately; this is done at a later stage.

The Carl row now is also expired and invisible for the active transactions. The second SELECT removed it from the City index chain; however, it is still present in the Name index chain and cannot be deallocated. Figure 9-3 shows the state of the data at this moment.

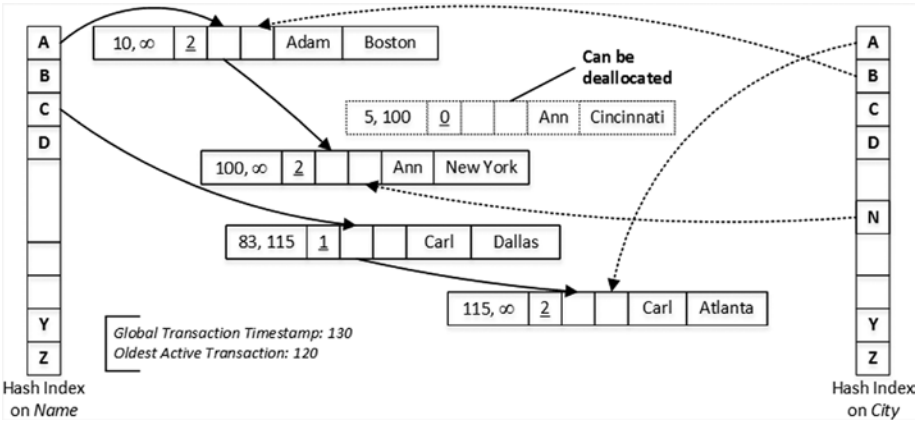


Figure 9-3. *State of the data after the second two queries*

■ **Important** You should remember that the *Oldest Active Transaction Timestamp* controls when expired rows can be removed from the index chains and deallocated. Long-running and abandon transactions can defer garbage collection and lead to a situation when the system runs out of memory due to an excessive number of expired rows.

When the transaction is complete, In-Memory OLTP places the information about it in the queue used by the idle worker thread, which is responsible for garbage collection management. The idle worker thread wakes up every minute or, in case of a heavy load, when the number of completed transactions exceeds the predefined threshold. It analyzes the list of completed transactions and the *Oldest Active Transaction Timestamp* in the system, and separates completed transactions to 16 different queues called *generations*, sorting them based on their *Global Transaction Timestamp* values.

- **Generation 0** contains the list of transactions that were completed earlier than the current *Oldest Active Transaction Timestamp*. Rows generated by those transactions are immediately available for the garbage collection.
- **Generations 1-14** stores the list of transactions that were completed after the current *Oldest Active Transaction Timestamp*. Each generation can hold information about up to 16 transactions. As you can guess, a system can hold up to 224 transactions in generations 1-14 queues.
- **Generation 15** stores the information about the remaining transactions completed after the current *Oldest Active Transaction Timestamp*. There is no limit on the number of transactions that can be stored there.

Every transaction in the queue exposes its *write set* to the idle worker thread, which builds the set of the 16-row *work items* for deallocation. Those work items are distributed across another set of *worker queues*-one queue per scheduler-and then they are picked up and processed by the user threads. The user threads pick up the items and perform deallocation after they complete their work on the other user transactions.

Figure 9-4 illustrates an example of the garbage collection workflow in a system that has an *Oldest Active Transaction Timestamp* of 10,000.

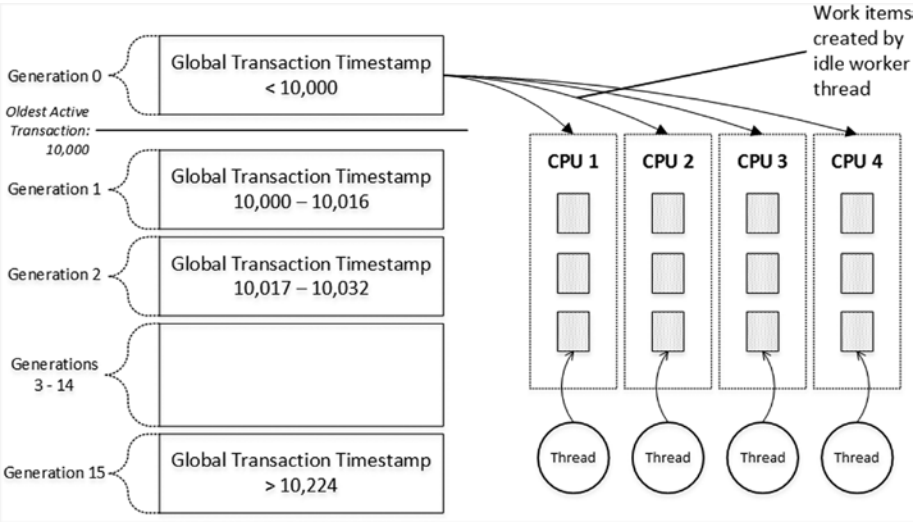


Figure 9-4. *Garbage Collection Workflow*

The user thread usually picks up the work items from the queue that belong to the same scheduler on which it is running. However, if the queue is empty, the thread checks the queues from the other CPUs that belong to the same NUMA node. Finally, in case of a heavy load in the system, the thread can pick up a work item from any queue, regardless of the NUMA node to which it belongs.

With the *hot* data and actively used indexes, user threads detect expired rows relatively quickly. However, with rarely used indexes and/or rarely accessed data, there is the possibility that expired rows may not be detected in a timely manner.

This is addressed by the idle worker thread, which periodically scans the indexes and detects expired rows there. The idle worker thread can either deallocate those rows immediately or add them to the work items after those rows have been unlinked from all index chains. This process is called a *dusty corners scan*.

As you can see, the garbage collection process in In-Memory OLTP is done asynchronously. Deleted rows and rows from aborted transactions continue to use system memory until they are deallocated. You need to remember this and reserve enough memory in the system to accommodate those rows.

Garbage Collection-Related Data Management Views

SQL Server exposes several data management views that can be used to monitor and analyze the garbage collection process.

- `sys.dm_xtp_gc_stats` provides statistics about the garbage collection process. It includes information about the number of rows examined by the garbage collection subsystem, the number of rows processed by user and idle worker threads, and quite a few other attributes. You can read more about this view at https://msdn.microsoft.com/en-us/library/dn133196.aspx.
- `sys.dm_xtp_gc_queue_stats` provides information about garbage collector worker queues. It provides information about total number of work items that were enqueued and dequeued, current queue length, last time the queue was accessed, and maximum depth the queue has seen. You can monitor the current queue length, making sure that the garbage collector is keeping up. More information is available at https://msdn.microsoft.com/en-us/library/dn268336.aspx.
- `sys.dm_db_xtp_gc_cycle_stats` provides information about the last (up to 1,024) garbage collection execution cycles including the time and duration of the cycle, and distribution of transactions between generations. You can use this view to find spikes in the garbage collection activity and during long-running transactions troubleshooting. You can read more about this view at https://msdn.microsoft.com/en-us/library/dn268337.aspx.
- Finally, `sys.dm_db_xtp_index_stats` includes several garbage collection-related metrics. The `rows_expired` column indicates how many rows have expired. `Rows_expired_removed` indicates the number of rows unlinked from the index chain. Phantom row columns provide information about rows inserted by aborted transactions. You can read more about this view at https://msdn.microsoft.com/en-us/library/dn133081.aspx.

Exploring the Garbage Collection Process

Let's examine the garbage collection process and its asynchronous nature. As the first step, create a memory-optimized table and populate it with 65,536 rows, as shown in Listing 9-3.

Listing 9-3. Table Creation

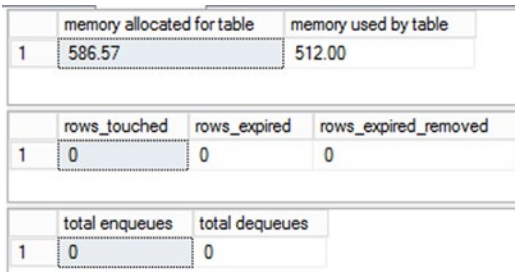
```
create table dbo.GCDemo
(
    ID int not null,
    Placeholder char(8000) not null,

    constraint PK_GCDemo
    primary key nonclustered hash(ID)
    with (bucket_count=16384),
)
with (memory_optimized=on, durability=schema_only)
go

;with N1(C) as (select 0 union all select 0) -- 2 rows
,N2(C) as (select 0 from N1 as t1 cross join N1 as t2) -- 4 rows
,N3(C) as (select 0 from N2 as t1 cross join N2 as t2) -- 16 rows
,N4(C) as (select 0 from N3 as t1 cross join N3 as t2) -- 256 rows
,N5(C) as (select 0 from N4 as t1 cross join N4 as t2) -- 65,536 rows
,Ids(Id) as (select row_number() over (order by (select null)) from N5)
insert into dbo.GCDemo(Id, Placeholder)
    select Id, Replicate('0',8000)
    from ids;
```

Let's look at amount of memory used in the table, index statistics, and garbage collection worker queues statistics using the code from Listing 9-4.

Listing 9-4. Analyzing Table Memory Usage, Index, and Worker Queues Statistics

```
select
    convert(decimal(7,2),memory_allocated_for_table_kb / 1024.)
                as [memory allocated for table]
    ,convert(decimal(7,2),memory_used_by_table_kb / 1024.)
                as [memory used by table]
from
    sys.dm_db_xtp_table_memory_stats
where
    object_id = object_id(N'dbo.GCDemo');

select rows_touched, rows_expired, rows_expired_removed
from sys.dm_db_xtp_index_stats
where object_id = object_id(N'dbo.GCDemo');

select
    sum(total_enqueues) as [total enqueues]
    ,sum(total_dequeues) as [total dequeues]
from
    sys.dm_xtp_gc_queue_stats
```

Figure 9-5 illustrates the output of the queries. As you can see, the table has about 586MB allocated and 512MB of used space. None of the rows have been deleted or touched (scanned). I also restarted my test server right before the test, so the garbage collection worker queues are empty.



	memory allocated for table	memory used by table
1	586.57	512.00

	rows_touched	rows_expired	rows_expired_removed
1	0	0	0

	total enqueues	total dequeues
1	0	0

Figure 9-5. *Memory and garbage collection statistics after table creation*

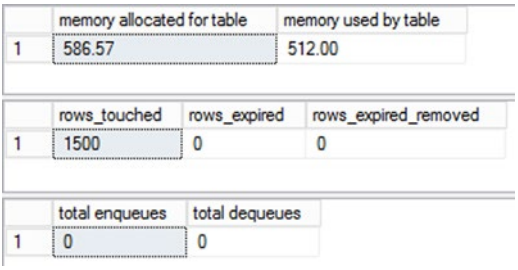
Let's run a few queries, analyzing the statistics after each run. In this book, I discuss results after each step; however, when you run this in your test environment, it is better to run all queries at once, persisting results in the temporary tables as is done in the script included with the companion materials of this book. This will help you to avoid the situation when idle worker threads start unexpectedly in the middle of execution.

As the first step, run the script that deletes 1,500 rows in the individual transactions (see Listing 9-5).

Listing 9-5. Deleting 1,500 Rows from the Table

```
declare
    @I int = 1

while @I <= 1500
begin
    delete from dbo.GCDemo where ID = @I;
    set @I += 1;
end;
```

Now run the code from Listing 9-4 again and look at the output. As you can see in Figure 9-6, index statistics indicate that the deletion statement touched 1,500 rows; however, none of them were marked as expired even though deletion statements ran in the individual autocommitted transactions.



	memory allocated for table	memory used by table
1	586.57	512.00

	rows_touched	rows_expired	rows_expired_removed
1	1500	0	0

	total enqueues	total dequeues
1	0	0

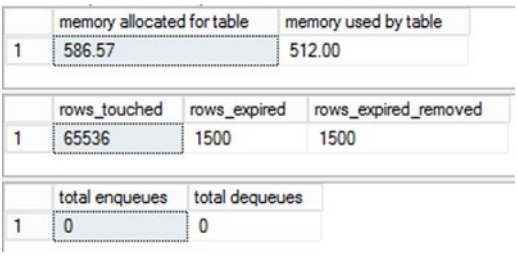
Figure 9-6. *Memory and garbage collection statistics after deletion*

As the next step, run a SELECT query that scans the entire index, as shown in Listing 9-6.

Listing 9-6. Scanning Table

```
select count(*) from dbo.GCDemo
```

Figure 9-7 illustrates the statistics after the scan. As you can see, In-Memory OLTP correctly identified rows as expired and unlinked them from the index row chains. However, none of the work items were enqueued in garbage collector worker items queues because the idle worker thread has not started yet.



	memory allocated for table	memory used by table
1	586.57	512.00

	rows_touched	rows_expired	rows_expired_removed
1	65536	1500	1500

	total enqueues	total dequeues
1	0	0

Figure 9-7. *Memory and garbage collection statistics after scan*

If you look at the statistics again after the idle worker thread execution, you will see the output shown in Figure 9-8. As you can see, the idle worker thread put items into the garbage collection worker queues where items are waiting for the user threads to deallocate them.

	memory allocated for table	memory used by table
1	586.57	512.00

	rows_touched	rows_expired	rows_expired_removed
1	65536	1500	1500

	total enqueues	total dequeues
1	93	0

Figure 9-8. *Memory and garbage collection statistics after the idle worker thread cycle*

If you scan the table with the query from Listing 9-6 again, you will see the statistics shown in Figure 9-9. A user thread processed and deallocated multiple items from the worker queues, releasing about 3MB of memory.

	memory allocated for table	memory used by table
1	586.54	509.18

	rows_touched	rows_expired	rows_expired_removed
1	129572	1500	1500

	total enqueues	total dequeues
1	93	23

Figure 9-9. *Memory and garbage collection statistics after the second scan*

The `sys.dm_db_xtp_gc_cycle_stats` view shows that the garbage collection idle worker thread performed just a handful of cycles (remember, I restarted SQL Server in my test environment before the test) and processed all completed transactions at once. You can see the partial output from the view in Figure 9-10.

	cycle_id	ticks_at_cycle_start	ticks_at_cycle_end	base_generation	xacts_copied_to_local
1	1	398260	398260	17	1
2	2	458265	458265	1521	1501
3	3	518270	518270	1521	0
4	4	578283	578283	1521	0

Figure 9-10. *Sys.dm_db_xtp_gc_cycle_stats view after the test*

The situation will change if you repeat entire test, deleting more rows from the table. The garbage collection process will be triggered based on the number of completed transactions in the queue rather than based on the timer.

Figure 9-11 shows the summary statistics from my environment when I repeated the test, deleting 32,768 rows in the individual transactions. Note that the garbage collection process was started at the middle of deletions rather than based on a timer.

	Stage	Alloc Memory	Used Memory	Touched	Expired	Removed	Enqueues	Dequeues
1	Initial	586.75	512.00	0	0	0	1	1
2	After Deletion	585.94	512.00	32768	0	0	2046	1
3	After Scan	585.10	448.45	65536	15398	15393	2046	512
4	After Delay	583.98	384.32	65536	15398	15393	2049	1024
5	After Second Scan	582.41	320.30	98304	15403	15397	2049	1536

Figure 9-11. *Memory and garbage collection statistics during the second set of tests*

You can also confirm it by looking at the `sys.dm_db_xtp_gc_cycle_stats` view output in Figure 9-12. It shows a much higher number of cycles with very short delays in between them.

	cycle_id	ticks_at_cycle_start	ticks_at_cycle_end	base_generation	xacts_copied_to_local
1	1	2807924	2807924	49	5
2	2	2839236	2839236	2049	1995
3	3	2839267	2839267	4097	2046
4	4	2839299	2839299	6145	2048
5	5	2839314	2839314	8193	2053
6	6	2839361	2839361	10241	2044
7	7	2839361	2839361	10273	33
8	8	2839392	2839392	12289	2014

Figure 9-12. *Sys.dm_db_xtp_gc_cycle_stats view after the second test*

Summary

The garbage collection process in In-Memory OLTP is designed to be non-blocking, cooperative, and scalable. Even though it is managed by a dedicated system thread (the idle worker thread) most of the work is done by the user threads. The idle worker thread wakes up every minute or when the number of completed transactions exceeds an internal threshold.

Deleted rows can be deallocated only after they are expired and their `EndTs` timestamp is older the than *Oldest Active Transaction Timestamp* in the system. Moreover, they need to be removed from all index row chains before deallocation. When user thread encounters an expired row, the thread unlinks it from the row chain. The idle worker thread periodically scans rarely accessed parts of the indexes during its *dusty corners scan* and processes expired rows that were missed by the user threads.

User threads provide information about completed transactions to the idle worker thread, which builds the list of work items that consist of 16-row batches to deallocate. The work items are distributed between garbage collector worker queues-one queue per scheduler in the system. In turn, user threads pickup one or several items from the worker queues and deallocate them.

Long-running and uncommitted transactions prevent rows from expiring by *freezing* the *Oldest Active Transaction Timestamp* in the system. This defers the garbage collection process and can lead to a situation where deleted rows use a large amount of memory.

CHAPTER 10

Deployment and Management

This chapter discusses the deployment and management aspects of systems that utilize In-Memory OLTP. It provides a set of guidelines about hardware and server configurations, and it covers In-Memory OLTP-related database administration and management tasks. Finally, this chapter gives an overview of the changes and enhancements in the catalog and data management objects related to In-Memory OLTP.

Hardware Considerations

In-Memory OLTP uses hardware in a different, and often more efficient, way than SQL Server Storage Engine. It is often possible to achieve high OLTP throughput even with mid-range servers. Moreover, In-Memory OLTP is highly scalable and it is possible to increase transaction throughput by adding more CPUs and memory to the server, and more drives to the disk array, as the load and amount of data in the system grows.

Obviously, you should not forget that In-Memory OLTP plays in the same sandbox with other SQL Server components, sharing resources with them. Memory becomes one of the most critical resources for which In-Memory OLTP and Storage Engines compete. The memory used by memory-optimized data is inaccessible to the Storage Engine and, therefore, cannot be used by the buffer pool. It is entirely possible that using In-Memory OLTP on servers with an insufficient amount of memory would degrade performance of the queries against on-disk tables if an excessive amount of physical I/O were required. You should remember this when designing the system and avoid putting unnecessary data into memory-optimized tables.

■ **Tip** Consider splitting *hot* and rarely accessed *historical* data between memory-optimized and on-disk tables. We will discuss this scenario in more depth in Chapter 11.

Let's discuss In-Memory OLTP requirements for different hardware components. Obviously, you need to take the workload from other SQL Server components into consideration when you build servers that utilize In-Memory OLTP.

CPU

The number of CPUs in the system greatly depends on the required OLTP throughput. However, as mentioned, it is entirely possible to achieve high transactional throughput even with a mid-range server. It is impossible to predict how many CPUs you will need without performing some testing and analysis; however, it is beneficial to use the proper hardware, which will allow you to scale and add more CPUs as load grows.

It is better to use Intel rather than AMD processors for OLTP workload even though some AMD processors have a lower SQL Server license cost. This situation may change in the future; however, as of summer 2015, Intel-based processors provide much better single-threaded performance, which is critical for In-Memory OLTP and OLTP workloads in general.

When possible, you should choose processors with a higher base clock speed. With SQL Server 2014 Enterprise Edition per-core licensing, you can often get a better OLTP performance/cost ratio by using high-end CPUs with a lower number of cores compared to slower CPUs with a higher number of cores.

Finally, you should have hyperthreading enabled on the servers. Microsoft states that hyperthreading can provide up to a 40 percent performance boost in some cases.

I/O Subsystem

As a general rule, you should place an In-Memory OLTP filegroup on the dedicated disk array optimized for sequential I/O performance. The sequential-only nature of In-Memory OLTP I/O patterns makes the choice between SSD- and magnetic media-based disk arrays more complicated. Even though solid state drives outperform magnetic media, high-performance magnetic media-based disk arrays can provide *good enough* sequential I/O performance to handle an In-Memory OLTP workload. Moreover, other factors, such as HBA and network bandwidth, can limit I/O throughput, making the disk performance difference negligible.

I/O read performance, however, is crucial at the database recovery stage. As you know, the In-Memory OLTP recovery process is highly scalable, with multiple schedulers loading data from the different checkpoint file pairs in parallel. Usually, I/O performance becomes the limiting factor in how fast SQL Server can recover memory-optimized data.

Recovery performance becomes even more important if a database has a low RTO metric (recovery time objective) in its SLA (service-level agreement). Even though databases with an In-Memory OLTP filegroup support piecemeal restore, SQL Server must bring all In-Memory OLTP data online together with the `PRIMARY` filegroup. You cannot postpone In-Memory OLTP filegroup recovery to a later stage in the restore.

One of the ways to improve recovery performance is to create multiple containers in the In-Memory OLTP filegroup, placing them in different disk arrays using different HBA adapters and, in the case of network storage, different access paths. SQL Server spreads checkpoint files across containers and will load them in parallel from multiple drives.

Listing 10-1 shows how to create a database with two containers in an In-Memory OLTP filegroup, placing them into the `H:\HKData` and `K:\HKData` folders, respectively.

Listing 10-1. Creating a Database with Two Containers in an In-Memory OLTP Filegroup

```
create database HKMultiContainers
on primary
(
    name = N'HKMultiContainers'
    ,filename = N'M:\HKMultiContainers.mdf'
),
filegroup HKData CONTAINS MEMORY_OPTIMIZED_DATA
(
    name = N'HKMultiContainers_HKData1'
    ,filename = N'H:\HKData\HKMultiContainers'
),
(
    name = N'HKMultiContainers_HKData2'
    ,filename = N'K:\HKData\HKMultiContainers'
)
log on
(
    name = N'HKMultiContainers_Log'
    ,filename = N'L:\KMultiContainers_log.ldf'
);
```

Continuous checkpoint and merge processes, on the other hand, do not usually put an extreme load on the disk subsystem. These processes utilize a streaming API and use a limited amount of threads to write data to the disk.

As for disk space, Microsoft recommends that you have enough space to accommodate 2X-3X of the size of the data from the durable memory-optimized tables. You should consider being closer to the higher mark to be on the safe side, especially if you expect your amount of data to grow.

Memory

You need to have enough memory in the system to accommodate the data from all of memory-optimized tables. SQL Server fails a transaction when it cannot allocate memory for the new row objects. Usually, SQL Server performs memory allocation during `INSERT` and `UPDATE` operations; however, a `DELETE` operation could also fail if a table has nonclustered indexes and there is not enough memory to accommodate new delta records or perform page merge operations.

Figure 10-1 shows an error message indicating an *out of memory* condition.

```
Msg 701, Level 17, State 103, Line 10
There is insufficient system memory in
resource pool 'default' to run this query.
```

Figure 10-1. *Out-of-memory error*

An out-of-memory situation essentially makes In-Memory OLTP data read-only. You can still query the data; however, you cannot perform any data modifications until the problem is resolved. When such conditions occur, it is beneficial to check the status of the garbage collection process to make sure that it has not been deferred by the old active transactions. We will discuss how to detect such transactions later in the chapter.

In a large number of cases, the only option to address an out-of-memory situation is to increase the amount of memory available to SQL Server and the In-Memory OLTP Engine. When this is impossible, you should detect the largest memory consumers in In-Memory OLTP and reduce their memory footprint by either refactoring or migrating them to on-disk tables. We will talk about how to detect them later in the chapter.

Estimating the Amount of Memory for In-Memory OLTP

Estimating the amount of memory required for memory-optimized tables is not a trivial task. As a rule of thumb, you can double the size of the data in the table as a basis for the estimation. For a more accurate estimate, however, you should factor the memory requirements for several different components:

- **Data rows** consist of a 24-byte header, an index pointer array (which is 8 bytes per index), and the payload (actual row data). For example, if your table has 1,000,000 rows and 3 indexes, and each row is about 200 bytes on average, you will need (24 + 3 * 8 + 200) * 1,000,000 = ~236.5MB of memory to store row data without any versioning overhead included in this number.
- **Hash indexes** use 8 bytes per bucket. If a table has two hash indexes defined with 1,500,000 buckets each, SQL Server will create indexes with 2,097,152 buckets, rounding the number of buckets specified in the index properties to next power of two. Those two indexes will use 2,097,152 * 2 * 8 = 32MB of memory.
- **Nonclustered index** memory usage is based on the number of unique index keys and index key size. If a table has a nonclustered index with 250,000 unique key values and each key value on average uses 30 bytes, it would use (30 + 8(pointer)) * 250,000 = ~9MB of memory. You can ignore the page header and non-leaf pages in your estimation as their sizes are insignificant compared to the leaf-level row size.
- **Row versioning** memory estimation depends on the duration of the longest transactions and the average number of data modifications (inserts and updates) per second. For example, if some processes in a system have 10-second transactions and, on average, the system handles 1,000 data modifications per second, you can estimate 10 * 1,000 * 248(row size) = ~2.4MB of memory for row versioning storage.

Obviously, these numbers outline the minimally required amount of memory. You should factor in future growth and changes in workload, and reserve some additional memory just to be safe.

As mentioned, it is also very important to remember that In-Memory OLTP does not work in vacuum; SQL Server needs to have enough memory available to the other components. Make sure to include this in your analysis.

You should also remember In-Memory OLTP memory requirements when you design High Availability and/or Disaster Recovery strategies in your system. It is not uncommon to see configurations where secondary and/or standby servers use less powerful hardware than the primary one. This approach helps to reduce hardware cost by allowing the system to operate with degraded performance in the event of a disaster.

You should be extremely careful with such an approach in case your database is using In-Memory OLTP technology. An insufficient amount of memory on secondary servers could break Always On synchronization and/or prevent you from restoring the database in the event of a disaster. The latter can also happen in scenarios when you want to bring the copy of the production database to development or testing environments where SQL Server does not have enough memory to accommodate In-Memory OLTP data from production.

Administration and Monitoring Tasks

Let's look at several common In-Memory OLTP-related database administration and monitoring tasks.

Limiting the Amount of Memory Available to In-Memory OLTP

SQL Server uses a *Resource Governor* to manage workload and system resource consumption. Internally, the Resource Governor uses *resource pools*, which represent a subset of the physical resources available to SQL Server. You can think about each resource pool as a virtual instance inside SQL Server, and you can control resources available to the resource pool by specifying its parameters. Finally, you can distribute the workload between resource pools or, to be precise, between resource pool *workgroups* using a *classification* process. Classification is done based on user-defined function, which allow you to define complex algorithms for such a purpose.

■ **Note** You can read more about the Resource Governor at
https://msdn.microsoft.com/en-us/bb933866.aspx.

Every Resource Governor configuration has two predefined resource pools created, *internal* and *default*. As you can guess by the name, the internal pool handles the internal SQL Server workload and the default pool handles the unclassified workload, which is all of the user workload that had not been classified to the other resource pools. You can create other resource pools as needed.

As mentioned, you can control CPU, memory, and I/O allocations between resource pools by specifying parameters, such as `MIN_CPU_PERCENT` and `MAX_CPU_PERCENT`, `MIN_MEMORY_PERCENT` and `MAX_MEMORY_PERCENT`, `AFFINITY` and a few others. You can bind a database to the resource pool, which, in the case of In-Memory OLTP, will allow you to limit the amount of memory for memory-optimized data in the system. Each database can be bound to a single resource pool; however, multiple databases can share the same pool. In this case, the limit would apply to all of them.

A resource pool can utilize up to 80 percent of the system memory, which sets the limit on the amount of memory available to In-Memory OLTP. That threshold guarantees that other SQL Server components have enough system memory to work and that the system remains stable under the memory pressure.

Listing 10-2 illustrates how to create and configure the resource pool, allowing it to use 40 percent of the system memory.

Listing 10-2. Creating a Resource Pool

```
create resource pool InMemoryDataPool
with
(
    min_memory_percent=40
    ,max_memory_percent=40
);

alter resource governor reconfigure;
```

When the resource pool is created, you can bind a database to it by using the `sys.sp_xtp_bind_db_resource_pool` stored procedure, as shown in Listing 10-3. Unfortunately, it does not automatically *transfer* previously allocated memory to the new pool so you need to take the database offline and bring it back online in order to do so. Remember that this leads to a recovery process, which can be time-consuming in the case of large amounts of In-Memory OLTP data.

Listing 10-3. Binding a Database to the Resource Pool

```
exec sys.sp_xtp_bind_db_resource_pool
    @database_name = 'InMemoryOLTPDemo'
    ,@pool_name = 'InMemoryDataPool';

-- You need to take DB offline and bring it
-- back online for the changes to take effect
alter database InMemoryOLTPDemo set offline;
alter database InMemoryOLTPDemo set online;
```

Similarly, you can remove the binding by calling the `sys.sp_xtp_unbind_db_resource_pool` stored procedure, as shown in Listing 10-4. The database will be bound back to the *default* resource pool after the call.

Listing 10-4. Removing the Binding Between a Database and a Resource Pool

```
exec sys.sp_xtp_unbind_db_resource_pool
    @database_name = 'InMemoryOLTPDemo';

-- You need to take DB offline and bring it
-- back online for the changes to take effect
alter database InMemoryOLTPDemo set offline;
alter database InMemoryOLTPDemo set online;
```

Monitoring Memory Usage for Memory-Optimized Tables

You can monitor memory usage of the various In-Memory OLTP objects by using a set of data management views along with the `Memory Usage by Memory Optimized Objects` report in SQL Server Management Studio.

The `sys.dm_db_xtp_table_memory_stats` view provides high-level memory usage statistics for the user and system memory-optimized tables in the current database. Listing 10-5 illustrates the query that uses this view.

Listing 10-5. Using sys.dm_db_xtp_table_memory_stats View

```
select
    ms.object_id
    ,s.name + '.' + t.name as [table]
    ,ms.memory_allocated_for_table_kb
    ,ms.memory_used_by_table_kb
    ,ms.memory_allocated_for_indexes_kb
    ,ms.memory_used_by_indexes_kb
from
    sys.dm_db_xtp_table_memory_stats ms
       left outer join sys.tables t on
           ms.object_id = t.object_id
       left outer join sys.schemas s on
           t.schema_id = s.schema_id
order by
    ms.memory_allocated_for_table_kb desc
```

Figure 10-2 shows the output of the query when I ran it against one of the databases. Rows with negative `object_id` belong to the system tables.

	object_id	table	memory_allocated_for_table_kb	memory_used_by_table_kb	memory_allocated_for_indexes_kb	memory_used_by_indexes_kb
1	277576027	Delivery.Orders	230578	229297	24576	24576
2	389576426	Delivery.Addresses	74633	74218	9216	9216
3	357576312	Delivery.Customers	7856	7812	7936	2743
4	485576768	Delivery.Rates	64	0	8	8
5	-2	NULL	13	13	16	16
6	517576882	Delivery.Drivers	6	6	648	29
7	549576996	Delivery.OrderStatuses	0	0	8	8
8	453576654	Delivery.RatePlans	0	0	2	2
9	421576540	Delivery.Services	0	0	2	2
10	-8	NULL	0	0	8	8
11	-7	NULL	0	0	32	32
12	-6	NULL	0	0	2	2
13	-5	NULL	0	0	24	24
14	-4	NULL	0	0	2	2
15	-3	NULL	0	0	2	2

Figure 10-2. *Output from sys.dm_db_xtp_table_memory_stats view*

■ **Note** You can read more about the `sys.dm_db_xtp_table_memory_stats` view at `https://msdn.microsoft.com/en-us/library/dn169142.aspx`.

The `sys.dm_db_xtp_memory_consumers` view provides information about database-level memory consumers. The `memory_consumer_type` column indicates the type of memory consumer in the output and can have one of three possible values:

- `VARHEAP (2)` indicates the database heap that is used to store user data and internal pages of nonclustered indexes.
- `HASH (3)` indicates memory used by the hash indexes.
- `PGPOOL (5)` shows the database page pool used by runtime operations. There is one memory consumer of such type per database.

You can use the `sys.dm_db_xtp_memory_consumers` view to track the memory allocation on a per-index basis, as shown in Listing 10-6.

Listing 10-6. Using sys.dm_db_xtp_memory_consumers View

```
select
    mc.object_id
    ,s.name + '.' + t.name as [table]
    ,i.name as [index]
    ,mc.memory_consumer_type_desc
    ,mc.memory_consumer_desc
    ,convert(decimal(9,3),mc.allocated_bytes / 1024. / 1024.)
        as [allocated (MB)]
    ,convert(decimal(9,3),mc.used_bytes / 1024. / 1024.)
        as [used (MB)]
    ,mc.allocation_count
```

```
from
    sys.dm_db_xtp_memory_consumers mc
       left outer join sys.tables t on
           mc.object_id = t.object_id
       left outer join sys.indexes i on
           mc.object_id = i.object_id and
           mc.index_id = i.index_id
       left outer join sys.schemas s on
           t.schema_id = s.schema_id
where -- Greater than 1MB
    mc.allocated_bytes > 1048576
order by
    [allocated (MB)] desc
```

Figure 10-3 shows the partial output of the query. Rows with a negative `object_id` belong to the system tables.

	object_id	table	index	memory_consumer_type_desc	memory_consumer_desc	allocated (MB)	used (MB)	allocation_count
1	NULL	NULL	NULL	VARHEAP	Database heap	305.375	304.039	1348685
2	389576426	Delivery.Addresses	PK__Addresse__091C2...	HASH	NULL	8.000	8.000	1
3	277576027	Delivery.Orders	PK__Orders__C3905BC...	HASH	NULL	8.000	8.000	1
4	277576027	Delivery.Orders	IDX_Orders_OrderNum	HASH	NULL	8.000	8.000	1
5	277576027	Delivery.Orders	IDX_Orders_CustomerId	HASH	NULL	8.000	8.000	1
6	357576312	Delivery.Customers	IDX_Customers	VARHEAP	Range index heap	6.750	1.679	868

Figure 10-3. *Output from sys.dm_db_memory_consumers view*

■ **Note** You can read more about the `sys.dm_db_xtp_memory_consumers` view at https://msdn.microsoft.com/en-us/library/dn133206.aspx.

The `sys.dm_xtp_system_memory_consumers` view provides information about memory used by system In-Memory OLTP components. Listing 10-7 illustrates the query that uses this view. Figure 10-4 shows the output of the query in my system.

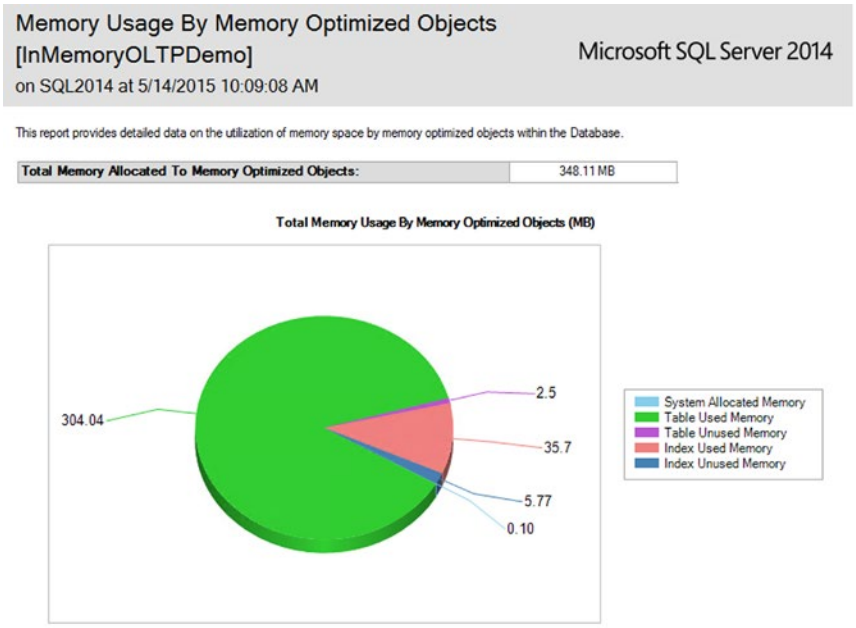
Listing 10-7. Using sys.dm_xtp_system_memory_consumers View

```
select
    memory_consumer_type_desc
    ,memory_consumer_desc
    ,convert(decimal(9,3),allocated_bytes / 1024. / 1024.)
        as [allocated (MB)]
    ,convert(decimal(9,3),used_bytes / 1024. / 1024.)
        as [used (MB)]
    ,allocation_count
from
   sys.dm_xtp_system_memory_consumers
order by
   [allocated (MB)] desc
```

	memory_consumer_type_desc	memory_consumer_desc	allocated (MB)	used (MB)	allocation_count
1	PGPOOL	System 256K page pool	392.250	392.250	1569
2	PGPOOL	System 4K page pool	47.438	47.438	12144
3	LOOKASIDE	Transaction write set	10.063	10.063	10068
4	VARHEAP	Lookaside heap	5.250	0.000	0
5	LOOKASIDE	Transaction constraint set	4.063	4.063	15213
6	VARHEAP	System heap	0.500	0.001	8
7	LOOKASIDE	Transaction	0.438	0.438	316
8	LOOKASIDE	Transaction partially-inserted rows set	0.188	0.188	366
9	LOOKASIDE	Hash cursor	0.188	0.188	744
10	LOOKASIDE	Transaction scan set	0.125	0.125	63
11	LOOKASIDE	Transaction read set	0.125	0.125	63
12	LOOKASIDE	Redo transaction map entry	0.125	0.125	1638
13	LOOKASIDE	Recovery table cache entry	0.125	0.125	4096
14	LOOKASIDE	Transaction recent rows	0.000	0.000	0
15	LOOKASIDE	Range cursor	0.000	0.000	0
16	LOOKASIDE	Sequence object insert row	0.000	0.000	0
17	LOOKASIDE	Sequence object map entry	0.000	0.000	0
18	LOOKASIDE	Sequence object values map	0.000	0.000	0
19	LOOKASIDE	GC transaction map entry	0.000	0.000	0
20	LOOKASIDE	Transaction dependent ring buffer	0.000	0.000	0
21	LOOKASIDE	Transaction save-point set entry	0.000	0.000	0
22	LOOKASIDE	Transaction save-point set	0.000	0.000	0
23	PGPOOL	System 64K page pool	0.000	0.000	0

Figure 10-4. *Output from sys.dm_xtp_system_memory_consumers view*

You can access the `Memory Usage by Memory Optimized Objects` report in the *Reports* ➤ *Standard Reports* section in the database context menu of the SQL Server Management Studio Object Explorer. Figure 10-5 illustrates the output of the report. As you can see, this report returns similar data to the `sys.dm_db_xtp_table_memory_stats` view.



Memory Usage Details for Memory Optimized Tables (MB)

Table Name	Table Used Memory	Table Unused Memory	Index Used Memory	Index Unused Memory
Services	0.00	0.00	0.00	0.00
Rates	0.00	0.06	0.01	0.00
OrderStatuses	0.00	0.00	0.01	0.00
Addresses	72.48	0.58	9.00	0.00
RatePlans	0.00	0.00	0.00	0.00
Drivers	0.01	0.00	0.03	0.60
Orders	223.92	1.80	24.00	0.00
Customers	7.63	0.06	2.65	5.17

Figure 10-5. *Memory Usage By Memory Optimized Objects report output*

Monitoring In-Memory OLTP Transactions

The `sys.dm_db_xtp_transactions` view provides information about active In-Memory OLTP transactions in the system. The most notable columns in the view are the following:

- `xtp_transaction_id` is the internal ID of the transaction in the In-Memory OLTP Transaction Manager.
- `transaction_id` is the transaction id in the system. You can use it in joins with other transaction management views, such as `sys.dm_tran_active_transactions`. In-Memory OLTP-only transactions, such as transactions started by natively compiled stored procedures, return `transaction_id` as 0.
- `session_id` indicates the session that started a transaction.

- `begin_tsn` and `end_tsn` indicate transaction timestamps.
- `state` and `state_desc` indicate the state of a transaction. The possible values are `(0)-ACTIVE`, `(1)-COMMITTED`, `(2)-ABORTED`, `(3)-VALIDATING`.
- `result` and `result_desc` indicate the result of a transaction. The possible values are `(0)-IN PROGRESS`, `(1)-SUCCESS`, `(2)-ERROR`, `(3)-COMMIT DEPENDENCY`, `(4)-VALIDATION FAILED (RR)` indicates repeatable read rules violation, `(5)-VALIDATION FAILED (SR)` indicates serializable rules violation, `(6)-ROLLBACK`.

You can use the `sys.dm_db_xtp_transactions` view to detect long-running and orphan transactions in the system. As you remember, these transactions can defer the garbage collection process and lead to out-of-memory errors.

Listing 10-8 shows a query that returns information about the five oldest active In-Memory OLTP transactions in the system.

Listing 10-8. Getting Information About the Five Oldest Active In-Memory OLTP Transactions

```
select top 5
    t.session_id
    ,t.transaction_id
    ,t.begin_tsn
    ,t.end_tsn
    ,t.state_desc
    ,t.result_desc
    ,substring(
        qt.text
        ,er.statement_start_offset / 2 + 1
        ,(case er.statement_end_offset
             when -1 then datalength(qt.text)
             else er.statement_end_offset
          end - er.statement_start_offset
        ) / 2 +1
    ) as SQL
from
    sys.dm_db_xtp_transactions t
        left outer join sys.dm_exec_requests er on
            t.session_id = er.session_id
        outer apply
            sys.dm_exec_sql_text(er.sql_handle) qt
where
    t.state in (0,3) /* ACTIVE/VALIDATING */
order by
    t.begin_tsn
```

Figure 10-6 illustrates the output of the query.

	session_id	xtp_transaction_id	transaction_id	begin_tsn	end_tsn	state_desc	result_desc	SQL
1	58	775	24730	55	0	ACTIVE	IN PROGRESS	NULL
2	54	742	23026	55	0	ACTIVE	IN PROGRESS	NULL
3	56	755	23776	55	0	ACTIVE	IN PROGRESS	NULL
4	57	757	23822	55	0	ACTIVE	IN PROGRESS	NULL
5	59	801	26413	56	0	ACTIVE	IN PROGRESS	select @Cnt = count(*) from Delivery.Or

Figure 10-6. *The five oldest active In-Memory OLTP transactions in the system*

■ **Note** You can read more about the `sys.dm_db_xtp_transactions` view at `https://msdn.microsoft.com/en-us/library/dn133194.aspx`.

Collecting Execution Statistics for Natively Compiled Stored Procedures

By default, SQL Server does not collect execution statistics for natively compiled stored procedures due to the performance impact it introduces. You can enable such a collection at the procedure level with `sys.sp_xtp_control_proc_exec_stats` and at the statement-level with `sys.sp_xtp_control_query_exec_stats` system stored procedures.

Both procedures accept a Boolean `@new_collection_value` parameter, which indicates if the collection needs to be enabled or disabled. In addition, `sys.sp_xtp_control_query_exec_stats` allows you to provide `@database_id` and `@object_id` to specify a stored procedure to monitor. It is also worth noting that SQL Server does not persist collection settings, and you will need to re-enable statistics collection after each SQL Server restart.

■ **Note** Execution statistics collection degrades the performance of the system. Do not collect execution statistics unless you are performing troubleshooting. Moreover, consider limiting collection to specific stored procedures to reduce the performance impact on the system.

When statistics have been collected, you can access them through the `sys.dm_exec_procedure_stats` and `sys.dm_exec_query_stats` views. Listing 10-9 shows the code that returns execution statistics for stored procedures using the `sys.dm_exec_procedure_stats` view. The code does not limit an output to natively compiled stored procedures; however, you can do it by joining `sys.dm_exec_procedure_stats` and `sys.sql_modules` views filtering by `uses_native_compliation = 1` value.

Listing 10-9. Analyzing Stored Procedures Execution Statistics

```
select
    object_name(object_id) as [Proc Name]
    ,execution_count as [Exec Cnt]
    ,total_worker_time as [Total CPU]
    ,convert(int,total_worker_time / 1000 / execution_count)
       as [Avg CPU] -- in Milliseconds
    ,total_elapsed_time as [Total Elps]
    ,convert(int,total_elapsed_time / 1000 / execution_count)
       as [Avg Elps] -- in Milliseconds
    ,cached_time as [Cached]
    ,last_execution_time as [Last Exec]
    ,sql_handle
    ,plan_handle
    ,total_logical_reads as [Reads]
    ,total_logical_writes as [Writes]
from
    sys.dm_exec_procedure_stats
order by
    [AVG CPU] desc
```

Figure 10-7 illustrates the output of the code from Listing 10-9. As you can see, neither the `sql_handle` nor `plan_handle` columns are populated. Execution plans for natively compiled stored procedures are embedded into the code and are not cached in the plan cache. Nor are I/O related statistics provided. Natively compiled stored procedures work with memory-optimized tables only, and therefore there is no I/O involved.

	Proc Name	Exec Cnt	Total CPU	Avg CPU	Total Elps	Avg Elps	Cached	Last Exec	sql_handle	plan_handle	Reads	Writes
1	InsertCustomers	2	6234000	3117	7069937	3534	2014-04-02 21:09...	2014-04-03 07:...	0x0000000000000000...	0x0000000000000000...	0	0
2	DeleteCustomers	2	328000	164	868088	434	2014-04-02 21:19...	2014-04-03 07:...	0x0000000000000000...	0x0000000000000000...	0	0

Figure 10-7. *Data from thesys.dm_exec_procedure_stats view*

Listing 10-10 shows the code that obtains execution statistics for individual statements using the `sys.dm_exec_query_stats` view.

Listing 10-10. Analyzing Stored Procedure Statement Execution Statistics

```
select
   substring(qt.text
      ,(qs.statement_start_offset/2)+1
      ,(case qs.statement_end_offset
           when -1 then datalength(qt.text)
           else qs.statement_end_offset
        end - qs.statement_start_offset) / 2 +1
   ) as SQL
```

```
   ,qs.execution_count as [Exec Cnt]
   ,qs.total_worker_time as [Total CPU]
   ,convert(int,qs.total_worker_time / 1000 /
     qs.execution_count) as [Avg CPU] -- In MS
   ,total_elapsed_time as [Total Elps]
   ,convert(int,qs.total_elapsed_time / 1000 /
     qs.execution_count) as [Avg Elps] -- In MS
   ,qs.creation_time as [Cached]
   ,last_execution_time as [Last Exec]
   ,qs.plan_handle
   ,qs.total_logical_reads as [Reads]
   ,qs.total_logical_writes as [Writes]
from
   sys.dm_exec_query_stats qs
      cross apply sys.dm_exec_sql_text(qs.sql_handle) qt
where
   qs.plan_generation_num is null
order by
   [AVG CPU] desc
```

Figure 10-8 illustrates the output of the code from Listing 10-10. Like procedure execution statistics, it is impossible to obtain the execution plans of the statements. However, you can analyze the CPU time consumed by individual statements and the frequency of their execution.

	SQL	Exec Cnt	Total CPU	Avg CPU	Total Elps	Avg Elps	Cached	Last Exec	plan_handle	Reads	Writes
1	delete from dbo.Custo...	2	218000	109	212116	106	2014-04-02 21:19:57...	2014-04-03 07:24:52...	0x000000000...	0	0
2	insert into dbo.Custom...	1999998	3174000	0	2348175	0	2014-04-02 21:09:11...	2014-04-03 07:24:56...	0x000000000...	0	0

Figure 10-8. *Data from the sys.dm_exec_query_stats view*

■ **Note** You can read more about the `sys.sp_xtp_control_proc_exec_stats` procedure at https://msdn.microsoft.com/en-us/library/dn435918.aspx. More information about the `sys.sp_xtp_control_query_exec_stats` procedure is available at https://msdn.microsoft.com/en-us/library/dn435917.aspx.

Metadata Changes and Enhancements

In-Memory OLTP introduces a large number of changes in catalog and data management views.

Catalog Views

In-Memory OLTP introduces the new catalog view `sys.hash_indexes`. As you can guess by the name, this view provides information about hash indexes defined in the database. It is inherited from and has the same columns as the `sys.indexes` view, adding one extra column called `bucket_count`. You can read about this view at https://msdn.microsoft.com/en-us/library/dn133205.aspx.

Other catalog view changes include the following:

- The `sys.tables` view has three new columns. The `Is_memory_optimized` column indicates if a table is memory-optimized. The `durability` and `durability_desc` columns indicate a durability mode for memory-optimized tables. The values are `(0)-SCHEMA_AND_DATA` and `(1)-SCHEMA_ONLY`.
- The `sys.indexes` view has a new possible value in the `type` and `type_description` columns, such as `(7)-NONCLUSTERED HASH`. Nonclustered Bw-Tree indexes use the value of `(2)-NONCLUSTERED` as the regular nonclustered B-Tree indexes defined on on-disk tables.
- The `sys.sql_modules` and `sys.all_sql_modules` have a new column called `uses_native_compilation`.
- The `sys.table_types` view has a new column called `is_memory_optimized`, which indicates if a type represents a memory-optimized table variable.
- The `sys.data_spaces` view now has a new `type` and `type_desc` value of `(FX)-MEMORY_OPTIMIZED_DATA_FILEGROUP`.

Data Management Views

In-Memory OLTP introduces a large set of new data management views, which can be easily detected by the `xtp_` prefix in their names. The naming convention also provides information about their scope. `Sys.dm_xtp_*` views return instance-level and `sys.dm_db_xtp_*` views provide database-level information. Let's look at them in more detail, grouping them by areas.

Object and Index Statistics

The following data management views provide index- and data modification-related statistics:

- `sys.dm_db_xtp_object_stats` reports the number of rows affected by data modification operations on a per-objects basis. You can use this view to analyze the volatility of the data from memory-optimized tables, correlating it with index usage statistics. As with on-disk tables, you can improve data modification performance by removing rarely used indexes defined on volatile tables. More information about this view is available at https://msdn.microsoft.com/en-us/library/dn133191.aspx.
- `sys.dm_db_xtp_index_stats` returns information about index usage, including data about expired rows. You can read about this view at https://msdn.microsoft.com/en-us/library/dn133081.aspx.
- `sys.dm_db_xtp_hash_index_stats` provides information about hash indexes, such as number of buckets in the index, number of empty buckets, and row chain length information. This view is useful when you need to analyze the state of hash indexes and fine-tune their `bucket_count` allocations. You can read about this view at https://msdn.microsoft.com/en-us/library/dn296679.aspx.

Listing 10-11 shows the script that you can use to find hash indexes with potentially suboptimal `bucket_count` value.

Listing 10-11. Obtaining Information About Hash Indexes with Potentially Suboptimal bucket_count Value

```
select
    s.name + '.' + t.name as [Table]
    ,i.name as [Index]
    ,stat.total_bucket_count as [Total Buckets]
    ,stat.empty_bucket_count as [Empty Buckets]
    ,floor(100. * empty_bucket_count / total_bucket_count)
        as [Empty Bucket %]
    ,stat.avg_chain_length as [Avg Chain]
    ,stat.max_chain_length as [Max Chain]
from
    sys.dm_db_xtp_hash_index_stats stat
        join sys.tables t on
            stat.object_id = t.object_id
        join sys.indexes i on
            stat.object_id = i.object_id and
            stat.index_id = i.index_id
```

```
        join sys.schemas s on
            t.schema_id = s.schema_id
where
    stat.avg_chain_length > 3 or
    stat.max_chain_length > 50 or
    floor(100. * empty_bucket_count /
        total_bucket_count) > 50
```

Memory Usage Statistics

We already discussed memory usage-related views in this chapter. However, as a quick overview, the views are the following:

- `sys.dm_xtp_system_memory_consumers` reports information about system-level memory consumers in the system. More information about this view is available at https://msdn.microsoft.com/en-us/library/dn133200.aspx.
- `sys.dm_db_xtp_table_memory_stats` provides memory usage statistics on per-object level. You can read more at https://msdn.microsoft.com/en-us/library/dn169142.aspx.
- `sys.dm_db_xtp_memory_consumers` provides information about database-level memory consumers. You can use this view to analyze per-index memory allocation in the system. The documentation is available at https://msdn.microsoft.com/en-us/library/dn133206.aspx.

Transaction Management

The following views provide transaction-related statistics in the system:

- `sys.dm_xtp_transaction_stats` reports statistics about transactional activity in the system since the last server restart. It includes the number of transactions, information about transaction log activity, and quite a few other metrics. More information about this view is available at https://msdn.microsoft.com/en-us/library/dn133198.aspx.
- `sys.dm_db_xtp_transactions` provides information about currently active transactions in the system. We discussed this view in this chapter and you can read more about it at https://msdn.microsoft.com/en-us/library/dn133194.aspx.

Garbage Collection

The following views provide information about garbage collection process in the system:

- `sys.dm_xtp_gc_stats` reports the overall statistics about the garbage collection process. More information is available at https://msdn.microsoft.com/en-us/library/dn133196.aspx.
- `sys.dm_xtp_gc_queue_stats` provides information about the state of garbage collection worker item queues. You can use this view to monitor if the garbage collection deallocation process is keeping up with the system load. You can read more about this view at https://msdn.microsoft.com/en-us/library/dn268336.aspx.
- `sys.dm_db_xtp_gc_cycle_stats` provides information about idle worker thread generation queues. We discussed this view in detail in Chapter 9 and you can read more about it at https://msdn.microsoft.com/en-us/library/dn268337.aspx.

Checkpoint

The following views provide information about checkpoint operations in the current database:

- `sys.dm_db_xtp_checkpoint_stats` reports the overall statistics about database checkpoint operations. It includes log file I/O statistics, amount of data processed during continuous checkpoint, time since last checkpoint operation, and quite a few other metrics. More information about this view is available at https://msdn.microsoft.com/en-us/library/dn133197.aspx.
- `sys.dm_db_xtp_checkpoint_files` provides information about checkpoint file pairs in the database. Appendix C shows this view in action and you can read more about it at https://msdn.microsoft.com/en-us/library/dn133201.aspx.
- `sys.dm_db_xtp_merge_requests` tracks checkpoint merge requests in the database. You can read more about it at https://msdn.microsoft.com/en-us/library/dn465868.aspx.

Extended Events and Performance Counters

SQL Server 2014 introduces three new xEvent packages that contain a large number of Extended Events. You can use the code from Listing 10-12 to get the list of Extended Events from those packages along with their descriptions.

Listing 10-12. Analyzing In-Memory OLTP Extended Events

```
select
    xp.name as [package]
    ,xo.name as [event]
    ,xo.description as [description]
from
    sys.dm_xe_packages xp
        join sys.dm_xe_objects xo on
            xp.guid = xo.package_guid
where
    xp.name like 'XTP%'
order by
    xp.name, xo.name
```

Figure 10-9 shows the partial output from the query.

	package	event	description
1	XtpCompile	cgen	Occurs at start of C code generation.
2	XtpCompile	invoke_cl	Occurs prior to the invocation of the C compiler.
3	XtpCompile	keyword_map	Event grouping keywords
4	XtpCompile	mat_export	Occurs at start of MAT export.
5	XtpCompile	pitgen_procs	Occurs at start of PIT generation for procedures.
6	XtpCompile	pitgen_tables	Occurs at start of PIT generation for tables.
7	XtpEngine	after_changestatetx_event	Fires after transaction changes state.
8	XtpEngine	alloctx_event	
9	XtpEngine	attempt_committx_event	Is raised when a transaction is asked to commit.
10	XtpEngine	before_changestatetx_event	Fires before transaction changes state.
11	XtpEngine	dependency_acquiredtx_event	Raised after transaction takes a dependency on a...
122	XtpEngine	xtp_merge_request_log_record	Indicates merge request log record is posted to the...
123	XtpEngine	xtp_merge_request_started	Indicates merge request has been picked up by th...
124	XtpEngine	xtp_root_deserialized	Indicates that the load of a checkpoint root is com...
125	XtpEngine	xtp_root_serialized	Indicates that the write of the checkpoint root is co...
126	XtpRuntime	bind_md	Occurs prior to binding metadata for a memory opti...
127	XtpRuntime	bind_tables	Occurs prior to binding tables for a natively compile...
128	XtpRuntime	create_table	Occurs prior to creating memory optimized table.
129	XtpRuntime	deserialize_md	Occurs prior to deserializing metadata.
130	XtpRuntime	keyword_map	Event grouping keywords
131	XtpRuntime	load_dll	Occurs prior to loading the generated DLL.
132	XtpRuntime	recover_done	Occurs at completion of checkpoint recovery of a ...

Figure 10-9. *In-Memory OLTP Extended Events*

Similarly, you can see the new performance counters with the query shown in Listing 10-13. Figure 10-10 shows a partial output of the query.

Listing 10-13. Analyzing In-Memory OLTP Performance Counters

```
select object_name, counter_name
from sys.dm_os_performance_counters
where object_name like 'XTP%'
order by object_name, counter_name
```

	object_name	counter_name
1	XTP Cursors	Cursor deletes/sec
2	XTP Cursors	Cursor inserts/sec
3	XTP Cursors	Cursor scans started/sec
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
23	XTP Garbage Collection	Sweep expiring rows touched/sec
24	XTP Garbage Collection	Sweep rows touched/sec
25	XTP Garbage Collection	Sweep scans started/sec
26	XTP Phantom Processor	Dusty corner scan retries/sec (Phantom-issued)
27	XTP Phantom Processor	Phantom expired rows removed/sec
28	XTP Phantom Processor	Phantom expired rows touched/sec
29	XTP Phantom Processor	Phantom expiring rows touched/sec
30	XTP Phantom Processor	Phantom rows touched/sec
31	XTP Phantom Processor	Phantom scans started/sec
32	XTP Storage	Checkpoints Closed
33	XTP Storage	Checkpoints Completed
34	XTP Storage	Core Merges Completed
35	XTP Storage	Merge Policy Evaluations
36	XTP Storage	Merge Requests Outstanding
37	XTP Storage	Merges Abandoned
38	XTP Storage	Merges Installed
39	XTP Storage	Total Files Merged
40	XTP Transaction Log	Log bytes written/sec
41	XTP Transaction Log	Log records written/sec
42	XTP Transactions	Cascading aborts/sec
43	XTP Transactions	Commit dependencies taken/sec
44	XTP Transactions	Read-only transactions prepared/sec
45	XTP Transactions	Save point refreshes/sec

Figure 10-10. *In-Memory OLTP Performance Counters*

■ **Note** You can read about In-Memory OLTP performance counters at https://msdn.microsoft.com/en-us/library/dn511015.aspx.

Summary

Choosing the right hardware is a crucial part of achieving good In-Memory OLTP performance and transactional throughput. In-Memory OLTP uses hardware in a different manner than the Storage Engine and you need to carefully plan the deployment and server configuration when a system uses In-Memory OLTP.

In-Memory OLTP benefits from single-threaded CPU performance. You should choose Intel-based CPUs with a high base clock speed and have hyperthreading enabled in the system.

You should store In-Memory OLTP checkpoint files in the disk array, which is optimized for sequential I/O performance. You can consider using multiple containers in an In-Memory OLTP filegroup, placing them on the different drives if database recovery time is critical.

Obviously, you should have enough memory in the system to accommodate In-Memory OLTP data, leaving enough memory for other SQL Server components. You can restrict In-Memory OLTP memory usage by configuring memory in the Resource Governor resource pool and binding the database there.

In-Memory OLTP provides a large set of data management views, performance counters, and Extended Events which you can use for system monitoring.

CHAPTER 11

Utilizing In-Memory OLTP

This chapter discusses several design considerations for systems utilizing In-Memory OLTP and shows a set of techniques that can be used to address some of In-Memory OLTP's limitations. Moreover, this chapter demonstrates how to benefit from In-Memory OLTP in scenarios when refactoring of existing systems is cost-ineffective. Finally, this chapter talks about systems with mixed workload patterns and how to benefit from the technology in those scenarios.

Design Considerations for the Systems Utilizing In-Memory OLTP

As with any new technology, adoption of In-Memory OLTP comes at a cost. You will need to acquire and/or upgrade to the Enterprise Edition of SQL Server 2014, spend time learning the technology, and, if you are migrating an existing system, refactor code and test the changes. It is important to perform a cost/benefits analysis and determine if In-Memory OLTP provides you with adequate benefits to outweigh the costs.

In-Memory OLTP is hardly a magical solution that will improve server performance by simply flipping a switch and moving data into memory. It is designed to address a specific set of problems, such as latch and lock contentions on very active OLTP systems. Moreover, it helps improve the performance of the small and frequently executed OLTP queries that perform point-lookups and small range scans.

In-Memory OLTP is less beneficial in the case of Data Warehouse systems with low concurrent activity, large amounts of data, and queries that require large scans and complex aggregations. While in some cases it is still possible to achieve performance improvements by moving data into memory, you can often obtain better results by implementing columnstore indexes, indexed views, data compression, and other database schema changes. It is also worth remembering that most performance improvements with In-Memory OLTP are achieved by using natively compiled stored procedures, which can rarely be used in Data Warehouse workloads due to the limited set of T-SQL features that they support.

The situation is more complicated with systems that have a mixed workload, such as an OLTP workload against *hot*, recent data and a Data Warehouse/Reporting workload against old, *historical* data. In those cases, you can partition the data into multiple tables,

moving recent data into memory and keeping old, historical data on-disk. Partition views can be beneficial in this scenario by hiding the storage details from the client applications. We will discuss such implementation later in this chapter.

Another important factor is whether you plan to use In-Memory OLTP during the development of new or the migration of existing systems. It is obvious that you need to make changes in existing systems, addressing the limitations of memory-optimized tables, such as missing support of triggers, foreign key constraints, check and unique constraints, calculated columns, and quite a few other restrictions.

There are other factors that can greatly increase migration costs. The first is the 8,060-byte maximum row size limitation in memory-optimized tables without any off-row data storage support. This limitation can lead to a significant amount of work when the existing active OLTP tables use LOB data types, such as `(n)varchar(max)`, `xml`, `geography` and a few others. While it is possible to change the data types, limiting the size of the strings or storing XML as text or in binary format, such changes are complex, time-consuming, and require careful planning. Don't forget that In-Memory OLTP does not allow you to create a table if there is a *possibility* that the size of a row exceeds 8,060 bytes. For example, you cannot create a table with three `varchar(3000)` columns even if you do not plan to exceed the 8,060-byte row size limit.

Indexing of memory-optimizing tables is another important factor. While nonclustered indexes can mimic some of the behavior of indexes in on-disk tables, there is still a significant difference between them. Nonclustered indexes are unidirectional, and they would not help much if the data needs to be accessed in the opposite sorting order of an index key. This often requires you to reevaluate your index strategy when a table is moved from disk into memory. However, the bigger issue with indexing is the requirement of case-sensitive binary collation of the indexed text columns. This is a breaking change in system behavior, and it often requires non-trivial changes in the code and some sort of data conversion.

It is also worth noting that using binary collations for data will lead to changes in the T-SQL code. You will need to specify collations for variables in stored procedures and other T-SQL routines, unless you change the database collation to be a binary one. However, if the database and server collations do not match, you will need to specify a collation for the columns in temporary tables created in `tempdb`.

There are plenty of other factors to consider. However, the key point is that you should perform a thorough analysis before starting a migration to In-Memory OLTP. Such a migration can have a very significant cost, and it should not be done unless it benefits the system.

SQL Server 2014 provides the tools that can help during In-Memory OLTP migration. These tools are based on the Management Data Warehouse, and they provide you with a set of data collectors and reports that can help identify the objects that would benefit the most from the migration. While those tools can be beneficial during the initial analysis stage, you should not make a decision based solely on their output. Take into account all of the other factors and considerations we have already discussed in this book.

■ **Note** We will discuss migration tools in detail in Appendix D.

New development, on the other hand, is a very different story. You can design a new system and database schema taking In-Memory OLTP limitations into account. It is also possible to adjust some functional requirements during the design phase. As an example, it is much easier to store data in a case-sensitive way from the beginning compared to changing the behavior of existing systems after they were deployed to production.

You should remember, however, that In-Memory OLTP is an Enterprise Edition feature, and it requires powerful hardware with a large amount of memory. It is an expensive feature due to its licensing costs. Moreover, it is impossible to "set it and forget it." Database professionals should actively participate in monitoring and system maintenance after deployment. They need to monitor system memory usage, analyze data and recreate hash indexes if bucket counts need to be adjusted, update statistics, redeploy natively compiled stored procedures, and perform other tasks as well.

All of that makes In-Memory OLTP a bad choice for Independent Software Vendors who develop products that need be deployed to a large number of customers. Moreover, it is not practical to support two versions of a system—with and without In-Memory OLTP—due to the increase in development and support costs.

Addressing In-Memory OLTP Limitations

Let's take a closer look at some of the In-Memory OLTP limitations and the ways to address them. Obviously, there is more than one way to skin a cat, and you can work around these limitations differently.

8,060-Byte Maximum Row Size Limit

The 8,060-byte maximum row size limit is, perhaps, one of the biggest roadblocks in widespread technology adoption. This limitation essentially prevents you from using `(max)` data types along with CLR and system data types that require off-row storage, such as `XML`, `geometry`, `geography` and a few others. Even though you can address this by changing the database schema and T-SQL code, these changes are often expensive and time-consuming.

When you encounter such a situation, you should analyze if LOB data types are required in the first place. It is not uncommon to see a column that never stores more than a few hundred characters defined as `(n)varchar(max)`. Consider an Order Entry system and `DeliveryInstruction` column in the `Orders` table. You can safely limit the size of the column to 500-1,000 characters without compromising the business requirements of the system.

Another example is a system that collects some semistructured sensor data from the devices and stores it in the `XML` column. If the amount of semistructured data is relatively small, you can store it in `varbinary(N)` column, which will allow you to move the table into memory.

■ **Tip** It is more efficient to use `varbinary` rather than `nvarchar` to store XML data in cases when you cannot use the `XML` data type.

Unfortunately, sometimes it is impossible to change the data types and you have to keep LOB columns in the tables. Nevertheless, you have a couple options to proceed.

The first approach is to split data between two tables, storing the key attributes in memory-optimized and rarely-accessed LOB attributes in on-disk tables. Again, consider the situation where you have an Order Entry system with the `Products` table defined as shown in Listing 11-1.

Listing 11-1. Products Table Definition

```
create table dbo.Products
(
    ProductId int not null identity(1,1),
    ProductName nvarchar(64) not null,
    ShortDescription nvarchar(256) not null,
    Description nvarchar(max) not null,
    Picture varbinary(max) null,

    constraint PK_Products
    primary key clustered(ProductId)
)
```

As you can guess, in this scenario, it is impossible to change the data types of the `Picture` and `Description` columns, which prevents you from making the `Products` table memory-optimized.

You can split that table into two, as shown in Listing 11-2. The `Picture` and `Description` columns are stored in an on-disk table while all other columns are stored in the memory-optimized table. This approach will improve performance for the queries against the `ProductsInMem` table and will allow you to access it from natively compiled stored procedures in the system.

Listing 11-2. Splitting Data Between Two Tables

```
create table dbo.ProductsInMem
(
    ProductId int not null identity(1,1)
        constraint PK_ProductsInMem
        primary key nonclustered hash
        with (bucket_count = 65536),
    ProductName nvarchar(64)
        collate Latin1_General_100_BIN2 not null,
    ShortDescription nvarchar(256) not null,

    index IDX_ProductsInMem_ProductName
    nonclustered(ProductName)
)
with (memory_optimized = on, durability = schema_and_data);
```

```
create table dbo.ProductAttributes
(
    ProductId int not null,
    Description nvarchar(max) not null,
    Picture varbinary(max) null,

    constraint PK_ProductAttributes
    primary key clustered(ProductId)
);
```

Unfortunately, it is impossible to define a foreign key constraint referencing a memory-optimized table, and you should support referential integrity in your code.

You can hide some of the implementation details from the `SELECT` queries by defining a view as shown in Listing 11-3. You can also define `INSTEAD OF` triggers on the view and use it as the target for data modifications; however, it is more efficient to update data in the tables directly.

Listing 11-3. Creating a View That Combines Data from Both Tables

```
create view dbo.Products(ProductId, ProductName,
    ShortDescription, Description, Picture)
as
    select
        p.ProductId, p.ProductName, p.ShortDescription
        ,pa.Description, pa.Picture
    from
        dbo.ProductsInMem p left outer join
            dbo.ProductAttributes pa on
                p.ProductId = pa.ProductId
```

As you should notice, the view is using an `outer join`. This allows SQL Server to perform join elimination when the client application does not reference any columns from the `ProductAttributes` table when querying the view. For example, if you ran the query from Listing 11-4, you would see the execution plan as shown in Figure 11-1. As you can see, there are no joins in the plan and the `ProductAttributes` table is not accessed.

Listing 11-4. Query Against the View

```
select ProductId, ProductName
from dbo.Products
```

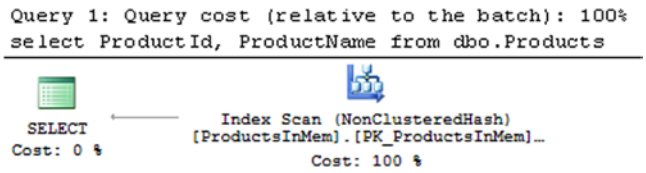


Figure 11-1. *Execution plan of the query*

You can use a different approach and store LOB data in memory-optimized tables, splitting it into multiple 8,000-byte chunks. Listing 11-5 shows the table that can be used for such a purpose.

Listing 11-5. Splitting LOB Data into Multiple Rows: Table Schema

```
create table dbo.LobData
(
    ObjectId int not null,
    PartNo smallint not null,
    Data varbinary(8000) not null,

    constraint PK_LobData
    primary key nonclustered hash(ObjectID, PartNo)
    with (bucket_count=1048576),

    index IDX_ObjectID
    nonclustered hash(ObjectId)
    with (bucket_count=1048576)

)
with (memory_optimized = on, durability = schema_and_data)
```

Listing 11-6 demonstrates how to insert `XML` data into the table using T-SQL code in interop mode. It uses an inline table-valued function called `dbo.SplitData` that accepts the `varbinary(max)` parameter and splits it into multiple 8,000-byte chunks.

Listing 11-6. Splitting LOB Data into Multiple Rows: Populating Data

```
create function dbo.SplitData
(
    @LobData varbinary(max)
)
returns table
as
return
```

```
(
    with Parts(Start, Data)
    as
    (
        select 1, substring(@LobData,1,8000)
        where @LobData is not null

        union all

        select
            Start + 8000
            ,substring(@LobData,Start + 8000,8000)
        from Parts
        where len(substring(@LobData,Start + 8000,8000)) > 0
    )
    select
        row_number() over(order by Start) as PartNo
        ,Data
    from
        Parts
)
go

declare
    @X xml

select @X =
    (select * from master.sys.objects for xml raw)

insert into dbo.LobData(ObjectId, PartNo, Data)
    select 1, PartNo, Data
    from dbo.SplitData(convert(varbinary(max),@X))
```

Figure 11-2 illustrates the contents of the LobData table after the insert.

	ObjectId	PartNo	Data
1	1	1	0xFFFE3C0072006F00770020006E0061006D0065003D0022...
2	1	2	0x45004D005F005400410042004C00450022002000630072 0...
3	1	3	0x690073005F006D0073005F00730068006900700070007000650...
4	1	4	0x7300790073007800 6D006900740071007500650075007500650...
5	1	5	0x2200340022002000700061007200650 06E0074005F006F0...
6	1	6	0x6F0062006A006500630074005F00690064003D002200300...
7	1	7	0x5200450044005F00500052004F00430045004400550052 0...

Figure 11-2. *Dbo.LobData table content*

■ **Note** SQL Server limits the CTE recursion level to 100 by default. You need to specify `OPTION (MAXRECURSION 0)` in the statement that uses the `SplitData` function in case of very large input.

You can construct original data using the code shown in Listing 11-7. Alternatively, you can develop a CLR aggregate and concatenate binary data there.

Listing 11-7. Splitting LOB Data into Multiple Rows: Getting Data

```
;with ConcatData(BinaryData)
as
(
    select
        convert(varbinary(max),
            (
                select convert(varchar(max),Data,2) as [text()]
                from dbo.LobData
                where ObjectId = 1
                order by PartNo
                for xml path('')
            ),2)
)
select convert(xml,BinaryData)
from ConcatData
```

The biggest downside of this approach is the inability to split and merge large objects in natively compiled stored procedures due to the missing `(max)` parameters and variables support. You should use the interop engine for this purpose. However, it is still possible to achieve performance improvements by moving data into memory even when the interop engine is in use.

This approach is also beneficial when memory-optimized tables are used just for the data storage, and all split and merge logic is done inside the client applications. We will discuss this implementation in much greater depth later in this chapter.

Lack of Uniqueness and Foreign Key Constraints

The inability to create unique and foreign key constraints rarely prevents us from adopting new technology. However, these constraints keep the data clean and allow us to detect data quality issues and bugs in the code at early stages of development.

Unfortunately, In-Memory OLTP does not allow you to define foreign keys or unique indexes and constraints besides a primary key. To make matter worse, the lock-free nature of In-Memory OLTP makes uniqueness support in the code tricky. In-Memory OLTP transactions do not see any uncommitted changes done by other transactions. For example, if you ran the code from Table 11-1 in the default `SNAPSHOT` isolation level, both transactions would successfully commit without seeing each other's changes.

Table 11-1. *Inserting the Duplicated Rows in the SNAPSHOT Isolation Level*

Session 1	Session 2
`set transaction isolation level snapshot`	
`begin tran`	
`if not exists (select * from dbo.ProductsInMem where ProductName = 'Surface 3')`	`set transaction isolation level snapshot begin tran`
`insert into dbo.ProductsInMem (ProductName) values ('Surface 3')`	`if not exists (select * from dbo.ProductsInMem where ProductName = 'Surface 3')`
`commit`	`insert into dbo.ProductsInMem (ProductName) values ('Surface 3')`
	`commit`

Fortunately, this situation can be addressed by using the `SERIALIZABLE` transaction isolation level. As you remember, In-Memory OLTP validates the serializable consistency rules by maintaining a transaction *scan set*. As part of the serializable rules validation at commit stage, In-Memory OLTP checks for *phantom rows*, making sure that other sessions do not insert any rows that were previously invisible to the transaction.

Listing 11-8 shows a natively compiled stored procedure that runs in the `SERIALIZABLE` isolation level and inserts a row into the `ProductsInMem` table we defined earlier. Any inserts done through this stored procedure guarantee uniqueness of the `ProductName` even in a multi-user concurrent environment.

The `SELECT` query builds a transaction scan set, which will be used for serializable rule validation. This validation will fail if any other session inserts a row with the same `ProductName` while the transaction is still active. Unfortunately, the first release of In-Memory OLTP does not support subqueries in natively compiled stored procedures and it is impossible to write the code using an `IF EXISTS` construct.

Listing 11-8. InsertProduct Stored procedure

```
create procedure dbo.InsertProduct
(
    @ProductName nvarchar(64) not null
    ,@ShortDescription nvarchar(256) not null
    ,@ProductId int output
)
with native_compilation, schemabinding, execute as owner
as
begin atomic with
(
    transaction isolation level = serializable
    ,language = N'English'
)
    declare
        @Exists bit = 0

    -- Building scan set and checking existense of the product
    select @Exists = 1
    from dbo.ProductsInMem
    where ProductName = @ProductName

    if @Exists = 1
    begin
       ;throw 50000, 'Product Already Exists', 1;
       return
    end

    insert into dbo.ProductsInMem(ProductName, ShortDescription)
    values(@ProductName, @ShortDescription);

    select @ProductID = scope_identity()
end
```

You can validate the behavior of the stored procedure by running it in two parallel sessions, as shown in Table 11-2. Session 2 successfully inserts a row and commits the transaction. Session 1, on the other hand, fails on commit stage with Error 41325.

Table 11-2. *Validating dbo.InsertProduct Stored Procedure*

Session 1	Session 2

`begin tran`
`  declare`
`    @ProductId int`
`  exec dbo.InsertProduct`
`    'Surface 3'`
`    ,'Microsoft Tablet'`
`    ,@ProductId output`
`commit`	

	`declare`
	`    @ProductId int`
	`exec dbo.InsertProduct`
	`  'Surface 3'`
	`  ,'Microsoft Tablet'`
	`  ,@ProductId output`
	`-- Executes and commits successfully`

`Error: Msg 41325, Level 16,`
`State 0, Line 62`
 	
`The current transaction failed`
`to commit due to a serializable`
`validation failure.`	

Obviously, this approach will work and enforce the uniqueness only when you have full control over the data access code in the system and have all `INSERT` and `UPDATE` operations performed through the specific set of stored procedures and/or code. The `INSERT` and `UPDATE` statements executed directly against a table could easily violate uniqueness rules. However, you can reduce the risk by revoking the `INSERT` and `UPDATE` permissions from users, giving them `EXECUTE` permission on the stored procedures instead.

You can use the same technique to enforce referential integrity rules. Listing 11-9 creates the `Orders` and `OrderLineItems` tables, and two stored procedures called `InsertOrderLineItems` and `DeleteOrders` enforce referential integrity between those tables there. I omitted the `OrderId` update scenario, which is very uncommon in the real world.

Listing 11-9. Enforcing Referential Integrity

```
create table dbo.Orders
(
    OrderId int not null identity(1,1)
        constraint PK_Orders
        primary key nonclustered hash
        with (bucket_count=1048576),
```

```
    OrderNum varchar(32)
        collate Latin1_General_100_BIN2 not null,
    OrderDate datetime2(0) not null
        constraint DEF_Orders_OrderDate
        default GetUtcDate(),
    /* Other Columns */
    index IDX_Orders_OrderNum
    nonclustered(OrderNum)
)
with (memory_optimized = on, durability = schema_and_data);

create table dbo.OrderLineItems
(
    OrderId int not null,
    OrderLineItemId int not null identity(1,1)
        constraint PK_OrderLineItems
        primary key nonclustered hash
        with (bucket_count=4194304),
    ArticleId int not null,
    Quantity decimal(8,2) not null,
    Price money not null,
    /* Other Columns */

    index IDX_OrderLineItems_OrderId
    nonclustered hash(OrderId)
    with (bucket_count=1048576)
)
with (memory_optimized = on, durability = schema_and_data);
go

create type dbo.tvpOrderLineItems as table
(
    ArticleId int not null
        primary key nonclustered hash
        with (bucket_count = 1024),
    Quantity decimal(8,2) not null,
    Price money not null
    /* Other Columns */
)
with (memory_optimized = on);
go

create proc dbo.DeleteOrder
(
    @OrderId int not null
)
```

```
with native_compilation, schemabinding, execute as owner
as
begin atomic
with
(
    transaction isolation level = serializable
    ,language=N'English'
)
    -- This stored procedure emulates ON DELETE NO ACTION
    -- foreign key constraint behavior
    declare
        @Exists bit = 0

    select @Exists = 1
    from dbo.OrderLineItems
    where OrderId = @OrderId

    if @Exists = 1
    begin
        ;throw 60000, N'Referential Integrity Violation', 1;
        return
    end

    delete from dbo.Orders where OrderId = @OrderId
end
go

create proc dbo.InsertOrderLineItems
(
    @OrderId int not null
    ,@OrderLineItems dbo.tvpOrderLineItems readonly
)
with native_compilation, schemabinding, execute as owner
as
begin atomic
with
(
    transaction isolation level = repeatable read
    ,language=N'English'
)
    declare
        @Exists bit = 0

    select @Exists = 1
    from dbo.Orders
    where OrderId = @OrderId
```

```
    if @Exists = 0
    begin
        ;throw 60001, N'Referential Integrity Violation', 1;
        return
    end

    insert into dbo.OrderLineItems(OrderId, ArticleId, Quantity, Price)
        select @OrderId, ArticleId, Quantity, Price
        from @OrderLineItems
end
```

It is worth noting that the `InsertOrderLineItems` procedure is using the `REPEATABLE READ` isolation level. In this scenario, you need to make sure that the referenced `Order` row has not been deleted during the execution and that `REPEATABLE READ` enforces this with less overhead than `SERIALIZABLE`.

Case-Sensitivity Binary Collation for Indexed Columns

As discussed, the requirement of having binary collation for the indexed text columns introduces a breaking change in the application behavior if case-insensitive collations were used before. Unfortunately, there is very little you can do about it. You can convert all the data and search parameters to uppercase or lowercase to address the situation; however, this is not always possible.

Another option is to store uppercase or lowercase data in another column, indexing and using it in the queries. Listing 11-10 shows such an example.

Listing 11-10. Storing Indexed Data in Another Column

```
create table dbo.Articles
(
    ArticleID int not null
        constraint PK_Articles
        primary key nonclustered hash
        with (bucket_count = 16384),
    ArticleName nvarchar(128) not null,
    ArticleNameUpperCase nvarchar(128)
        collate Latin1_General_100_BIN2 not null,
    -- Other Columns
    index IDX_Articles_ArticleNameUpperCase
    nonclustered(ArticleNameUpperCase)
);

-- Example of the query that uses upper case column
select ArticleId, ArticleName
from dbo.Articles
where ArticleNameUpperCase = upper(@ArticleName);
```

Unfortunately, memory-optimized tables don't support calculated columns and you will need to maintain the data in both columns manually in the code.

However, in the grand scheme of things, binary collations have benefits. The comparison operations on the columns that store data in binary collations are much more efficient compared to non-binary counterparts. You can achieve significant performance improvements when a large number of rows need to be processed.

One such example is a substring search in large tables. Consider the situation when you need to search by part of the product name in a large `Products` table. Unfortunately, a substring search will lead to the following predicate `WHERE ProductName LIKE '%' + @Param + '%'`, which is not *SARGable*, and SQL Server cannot use an *Index Seek* operation in such a scenario. The only option is to scan the data, evaluating every row in the table, which is significantly faster with binary collation.

Let's look at an example and create the table shown in Listing 11-11. The table has four text columns that store Unicode and non-Unicode data in binary and non-binary format. Finally, we populate it with 65,536 rows of random data.

Listing 11-11. Binary Collation Performance: Table Creation

```
create table dbo.CollationTest
(
    ID int not null,
    VarCol varchar(108) not null,
    NVarCol nvarchar(108)  not null,
    VarColBin varchar(108)
        collate Latin1_General_100_BIN2 not null,
    NVarColBin nvarchar(108)
        collate Latin1_General_100_BIN2 not null,

    constraint PK_CollationTest
    primary key nonclustered hash(ID)
    with (bucket_count=131072)
)
with (memory_optimized=on, durability=schema_only);

create table #CollData
(
    ID int not null,
    Col1 uniqueidentifier not null
        default NEWID(),
    Col2 uniqueidentifier not null
        default NEWID(),
    Col3 uniqueidentifier not null
        default NEWID()
);
```

```
;with N1(C) as (select 0 union all select 0) -- 2 rows
,N2(C) as (select 0 from N1 as T1 cross join N1 as T2) -- 4 rows
,N3(C) as (select 0 from N2 as T1 cross join N2 as T2) -- 16 rows
,N4(C) as (select 0 from N3 as T1 cross join N3 as T2) -- 256 rows
,N5(C) as (select 0 from N4 as T1 cross join N4 as T2) -- 65,536 rows
,IDs(ID) as (select row_number() over (order by (select NULL)) from N5)
insert into #CollData(ID)
    select ID from IDs;

insert into dbo.CollationTest(ID,VarCol,NVarCol,VarColBin,NVarColBin)
    select
        ID
        /* VarCol */
        ,convert(varchar(36),Col1) + convert(varchar(36),Col2) +
        convert(varchar(36),Col3)
        /* NVarCol */
        ,convert(nvarchar(36),Col1) + convert(nvarchar(36),Col2) +
        convert(nvarchar(36),Col3)
        /* VarColBin */
        ,convert(varchar(36),Col1) + convert(varchar(36),Col2) +
        convert(varchar(36),Col3)
        /* NVarColBin */
        ,convert(nvarchar(36),Col1) + convert(nvarchar(36),Col2) +
        convert(nvarchar(36),Col3)
    from
        #CollData
```

As the next step, run queries from Listing 11-12, comparing the performance of a search in different scenarios. All of the queries scan primary key hash index, evaluating the predicate for every row in the table.

Listing 11-12. Binary Collation Performance: Test Queries

```
declare
    @Param varchar(16)
    ,@NParam varchar(16)

-- Getting substring for the search
select
    @Param = substring(VarCol,43,6)
    ,@NParam = substring(NVarCol,43,6)
from
    dbo.CollationTest
where
    ID = 1000;
```

```
select count(*)
from dbo.CollationTest
where VarCol like '%' + @Param + '%';

select count(*)
from dbo.CollationTest
where NVarCol like '%' + @NParam + N'%';

select count(*)
from dbo.CollationTest
where VarColBin like '%' + upper(@Param) + '%'
            collate Latin1_General_100_Bin2;

select count(*)
from dbo.CollationTest
where NVarColBin like '%' + upper(@NParam) + N'%'
            collate Latin1_General_100_Bin2;
```

The execution time of all queries in my system are shown in Table 11-3. As you can see, the queries against binary collation columns are significantly faster, especially in the case of Unicode data.

Table 11-3. *Binary Collation Performace: Test Results*

varchar column with non-binary collation	varchar column with binary collation	nvarchar column with non-binary collation	nvarchar column with binary collation
191ms	109ms	769ms	62ms

Finally, it is worth noting that this behavior is not limited to memory-optimized tables. You will get a similar level of performance improvement with on-disk tables when binary collations are used.

Thinking Outside the In-Memory Box

Even though the limitations of the first release of In-Memory OLTP can make refactoring an existing systems cost-ineffective, you can still benefit from it by using some In-Memory OLTP components.

Importing Batches of Rows from Client Applications

In Chapter 12 of my book *Pro SQL Server Internals*, I compare the performance of several methods that inserted a batch of rows from the client application. I looked at the performance of calling individual `INSERT` statements; encoding the data into XML and passing it to a stored procedure; using the .Net `SqlBulkCopy` class; and passing data to a

stored procedure utilizing table-valued parameters. Table-valued parameters became the clear winner of the tests, providing performance on par with the `SqlBulkCopy` implementation plus the flexibility of using stored procedures during the import. Listing 11-13 illustrates the database schema and stored procedure I used in the tests.

Listing 11-13. Importing a Batch of Rows: Table, TVP, and Stored Procedure

```
create table dbo.Data
(
    ID int not null,
    Col1 varchar(20) not null,
    Col2 varchar(20) not null,
    /* Seventeen more columns Col3 - Col19*/
    Col20 varchar(20) not null,

    constraint PK_DataRecords
    primary key clustered(ID)
)
go

create type dbo.tvpData as table
(
    ID int not null,
    Col1 varchar(20) not null,
    Col2 varchar(20) not null,
    /* Seventeen more columns: Col3 - Col19 */
    Col20 varchar(20) not null,

    primary key(ID)
)
go

create proc dbo.InsertDataTVP
(
    @Data dbo.tvpData readonly
)
as
    insert into dbo.Data
    (
        ID,Col1,Col2,Col3,Col4,Col5,Col6,Col7
        ,Col8,Col9,Col10,Col11,Col12,Col13,Col14
        ,Col15,Col16,Col17,Col18,Col19,Col20
    )
        select ID,Col1,Col2,Col3,Col4,Col5,Col6
            ,Col7,Col8,Col9,Col10,Col11,Col12
            ,Col13,Col14,Col15,Col16,Col17,Col18
            ,Col19,Col20
        from @Data;
```

Listing 11-14 shows the ADO.Net code that performed the import in case of table-valued parameter.

Listing 11-14. Importing a Batch of Rows: Client Code

```
using (SqlConnection conn = GetConnection())
{
    /* Creating and populating DataTable object with dummy data */
    DataTable table = new DataTable();
    table.Columns.Add("ID", typeof(Int32));
    for (int i = 1; i <= 20; i++)
        table.Columns.Add("Col" + i.ToString(), typeof(string));
    for (int i = 0; i < packetSize; i++)
        table.Rows.Add(i, "Parameter: 1"
            ,"Parameter: 2"
            /* Other columns */
            ,"Parameter: 20");

    /* Calling SP with TVP parameter */
    SqlCommand insertCmd =
        new SqlCommand("dbo.InsertDataTVP", conn);
    insertCmd.Parameters.Add("@Data", SqlDbType.Structured);
    insertCmd.Parameters[0].TypeName = "dbo.tvpData";
    insertCmd.Parameters[0].Value = table;
    insertCmd.ExecuteNonQuery();
}
```

You can improve performance even further by replacing the `dbo.tvpData` table-valued type to be memory-optimized, which is transparent to the stored procedure and client code. Listing 11-15 shows the new type definition.

Listing 11-15. Importing a Batch of Rows: Defining a Memory-Optimized Table Type

```
create type dbo.tvpData as table
(
    ID int not null,
    Col1 varchar(20) not null,
    Col2 varchar(20) not null,
    /* Seventeen more columns: Col3 - Col19 */
    Col20 varchar(20) not null,

    primary key nonclustered hash(ID)
    with (bucket_count=65536)
)
with (memory_optimized=on);
```

The degree of performance improvement depends on the table schema, and it grows with the size of the batch. In my test environment, I got about 5-10 percent improvement on the small 5,000-row batches, 20-25 percent improvement on the 50,000-row batches, and 45-50 percent improvement on the 500,000-row batches.

You should remember, however, that memory-optimized table types cannot spill to `tempdb`, which can be dangerous in case of very large batches and with servers with an insufficient amount of memory. You should also define the `bucket_count` for the primary key based on the typical batch size, as discussed in Chapter 4 of this book.

■ **Note** You can download the test application from this book's companion materials and compare the performance of the various import methods.

Using Memory-Optimized Objects as Replacements for Temporary and Staging Tables

Memory-optimized tables and table variables can be used as replacements for on-disk temporary and staging tables. However, the level of performance improvement may vary, and it greatly depends on the table schema, workload patterns, and amount of data in the table.

Let's look at a few examples and, first, compare the performance of a memory-optimized table variable with on-disk temporary objects in a simple scenario, which you will often encounter in OLTP systems. Listing 11-16 shows stored procedures that insert up to 256 rows into the object, scanning it afterwards.

Listing 11-16. Comparing Performance of a Memory-Optimized Table Variable with On-Disk Temporary Objects

```
create type dbo.ttTemp as table
(
    Id int not null
        primary key nonclustered hash
        with (bucket_count=512),
    Placeholder char(255)
)
with (memory_optimized=on)
go

create proc dbo.TestInMemTempTables(@Rows int)
as
    declare
        @ttTemp dbo.ttTemp
        ,@Cnt int
```

```
    ;with N1(C) as (select 0 union all select 0) -- 2 rows
    ,N2(C) as (select 0 from N1 as t1 cross join N1 as t2) -- 4 rows
    ,N3(C) as (select 0 from N2 as t1 cross join N2 as t2) -- 16 rows
    ,N4(C) as (select 0 from N3 as t1 cross join N3 as t2) -- 256 rows
    ,Ids(Id) as (select row_number() over (order by (select null)) from N4)
    insert into @ttTemp(Id)
        select Id from Ids where Id <= @Rows;

    select @Cnt = count(*) from @ttTemp
go

create proc dbo.TestTempTables(@Rows int)
as
    declare
        @Cnt int

    create table #TTTemp
    (
        Id int not null primary key,
        Placeholder char(255)
    )

    ;with N1(C) as (select 0 union all select 0) -- 2 rows
    ,N2(C) as (select 0 from N1 as t1 cross join N1 as t2) -- 4 rows
    ,N3(C) as (select 0 from N2 as t1 cross join N2 as t2) -- 16 rows
    ,N4(C) as (select 0 from N3 as t1 cross join N3 as t2) -- 256 rows
    ,Ids(Id) as (select row_number() over (order by (select null)) from N4)
    insert into #TTTemp(Id)
        select Id from Ids where Id <= @Rows;

    select @Cnt = count(*) from #TTTemp
go

create proc dbo.TestTempVars(@Rows int)
as
    declare
        @Cnt int

    declare
        @ttTemp table
        (
            Id int not null primary key,
            Placeholder char(255)
        )
```

```
    ;with N1(C) as (select 0 union all select 0) -- 2 rows
    ,N2(C) as (select 0 from N1 as t1 cross join N1 as t2) -- 4 rows
    ,N3(C) as (select 0 from N2 as t1 cross join N2 as t2) -- 16 rows
    ,N4(C) as (select 0 from N3 as t1 cross join N3 as t2) -- 256 rows
    ,Ids(Id) as (select row_number() over (order by (select null)) from N4)
    insert into @ttTemp(Id)
        select Id from Ids where Id <= @Rows;

    select @Cnt = count(*) from @ttTemp
go
```

Table 11-4 illustrates the execution time of the stored procedures called 10,000 times in the loop. As you can see, the memory-optimized table variable outperformed on-disk objects. The level of performance improvements growth with the amount of data when on-disk tables need to allocate more data pages to store it.

Table 11-4. *Execution Time of Stored Procedures (10,000 Executions)*

	16 rows	64 rows	256 rows
Memory-Optimized Table Variable	920ms	1,496ms	3,343ms
Table Variable	1,203ms	2,994ms	8,493ms
Temporary Table	5,420ms	7,270ms	13,356ms

It is also worth mentioning that performance improvements can be even more significant in the systems with a heavy concurrent load due to possible allocation pages contention in `tempdb`.

You should remember that memory-optimized table variables do not keep index statistics, similar to on-disk table variables. The Query Optimizer generates execution plans with the assumption that they store just the single row. This cardinality estimation error can lead to highly inefficient plans, especially when a large amount of data and joins are involved.

■ **Important** As the opposite of on-disk table variables, statement-level recompile with `OPTION (RECOMPILE)` does not allow SQL Server to obtain the number of rows in memory-optimized table variables. The Query Optimizer always assumes that they store just a single row.

Memory-optimized tables can be used as the staging area for ETL processes. As a general rule, they outperform on-disk tables in `INSERT` performance, especially if you are using user database and durable tables for the staging.

Scan performance, on the other hand, greatly depends on the row size and number of data pages in on-disk tables. Traversing memory pointers is a fast operation and it is significantly faster compared to getting a page from the buffer pool. However, on-page

row access could be faster than traversing long memory pointers chain. It is possible that with the small data rows and large number of rows per page, on-disk tables would outperform memory-optimized tables in the case of scans.

Query parallelism is another important factor to consider. The first release of In-Memory OLTP does not support parallel execution plans. Therefore, large scans against on-disk tables could be significantly faster when they use parallelism.

Update performance depends on the number of indexes in memory-optimized tables, along with update patterns. For example, page splits in on-disk tables significantly decrease the performance of update operations.

Let's look at a few examples based on a simple ETL process that inserts data into an imaginary Data Warehouse with one fact, `FactSales`, and two dimension, the `DimDates` and `DimProducts` tables. The schema is shown in Listing 11-17.

Listing 11-17. ETL Performance: Data Warehouse Schema

```
create table dw.DimDates
(
    ADateId int identity(1,1) not null,
    ADate date not null,
    ADay tinyint not null,
    AMonth tinyint not null,
    AnYear smallint not null,
    ADayOfWeek tinyint not null,

    constraint PK_DimDates
    primary key clustered(ADateId)
);

create unique nonclustered index IDX_DimDates_ADate
on dw.DimDates(ADate);

create table dw.DimProducts
(
    ProductId int identity(1,1) not null,
    Product nvarchar(64) not null,
    ProductBin nvarchar(64)
        collate Latin1_General_100_BIN2
        not null,

    constraint PK_DimProducts
    primary key clustered(ProductId)
);

create unique nonclustered index IDX_DimProducts_Product
on dw.DimProducts(Product);
```

```
create unique nonclustered index IDX_DimProducts_ProductBin
on dw.DimProducts(ProductBin);

create table dw.FactSales
(
    ADateId int not null,
    ProductId int not null,
    OrderId int not null,
    OrderNum varchar(32) not null,
    Quantity decimal(9,3) not null,
    UnitPrice money not null,
    Amount money not null,

    constraint PK_FactSales
    primary key clustered(ADateId,ProductId,OrderId),

    constraint FK_FactSales_DimDates
    foreign key(ADateId)
    references dw.DimDates(ADateId),

    constraint FK_FactSales_DimProducts
    foreign key(ProductId)
    references dw.DimProducts(ProductId)
);
```

Let's compare the performance of two ETL processes utilizing on-disk and memory-optimized tables as the staging areas. We will use another table called `InputData` with 1,650,000 rows as the data source to reduce import overhead so we can focus on the `INSERT` operation performance. Listing 11-18 shows the code of the ETL processes.

Listing 11-18. ETL Performance: ETL Process

```
create table dw.FactSalesETLDisk
(
    OrderId int not null,
    OrderNum varchar(32) not null,
    Product nvarchar(64) not null,
    ADate date not null,
    Quantity decimal(9,3) not null,
    UnitPrice money not null,
    Amount money not null,
    /* Optional Placeholder Column */
    -- Placeholder char(255) null,
    primary key (OrderId, Product)
)
go
```

```
create table dw.FactSalesETLMem
(
    OrderId int not null,
    OrderNum varchar(32) not null,
    Product nvarchar(64)
        collate Latin1_General_100_BIN2 not null,
    ADate date not null,
    Quantity decimal(9,3) not null,
    UnitPrice money not null,
    Amount money not null,
    /* Optional Placeholder Column */
    -- Placeholder char(255) null,

    constraint PK_FactSalesETLMem
    primary key nonclustered hash(OrderId, Product)
    with (bucket_count = 2000000)

    /* Optional Index */
    -- index IDX_Product nonclustered(Product)
)
with (memory_optimized=on, durability=schema_and_data)
go

/*** ETL Process ***/

/* On Disk Table */

-- Step 1: Staging Table Insert
insert into dw.FactSalesETLDisk
    (OrderId,OrderNum,Product,ADate
        ,Quantity,UnitPrice,Amount)
        select OrderId,OrderNum,Product,ADate
            ,Quantity,UnitPrice,Amount
        from dbo.InputData;

/* Optional Index Creation */
--create index IDX1 on dw.FactSalesETLDisk(Product);

-- Step 2: DimProducts Insert
insert into dw.DimProducts(Product)
    select distinct f.Product
    from dw.FactSalesETLDisk f
    where not exists
        (
            select *
            from dw.DimProducts p
            where p.Product = f.Product
        );
```

```
-- Step 3: FactSales Insert
insert into dw.FactSales(ADateId,ProductId,OrderId,OrderNum,
    Quantity,UnitPrice,Amount)
        select d.ADateId,p.ProductId,f.OrderId,f.OrderNum,
            f.Quantity,f.UnitPrice,f.Amount
        from
            dw.FactSalesETLDisk f join dw.DimDates d on
                f.ADate = d.ADate
            join dw.DimProducts p on
                f.Product = p.Product;

/* Memory-Optimized Table */

-- Step 1: Staging Table Insert
insert into dw.FactSalesETLMem
    (OrderId,OrderNum,Product,ADate
        ,Quantity,UnitPrice,Amount)
        select OrderId,OrderNum,Product,ADate
            ,Quantity,UnitPrice,Amount
        from dbo.InputData;

-- Step 2: DimProducts Insert
insert into dw.DimProducts(Product)
    select distinct f.Product
    from dw.FactSalesETLMem f
    where not exists
        (
            select *
            from dw.DimProducts p
            where f.Product = p.ProductBin
        );

-- Step 3: FactSales Insert
insert into dw.FactSales(ADateId,ProductId,OrderId,OrderNum,
    Quantity,UnitPrice,Amount)
        select d.ADateId,p.ProductId,f.OrderId,f.OrderNum,
            f.Quantity,f.UnitPrice,f.Amount
        from
            dw.FactSalesETLMem f join dw.DimDates d on
                f.ADate = d.ADate
            join dw.DimProducts p on
                f.Product = p.ProductBin;
```

I have repeated the tests in four different scenarios, varying row size, with and without `Placeholder` columns and the existence of nonclustered indexes on `Product` columns. Table 11-5 illustrates the average execution time in my environment for the scenarios when tables don't have nonclustered indexes. Table 11-6 illustrates the scenario with additional nonclustered indexes on the `Product` column.

Table 11-5. *Execution Time of the Tests: No Additional Indexes*

	On-Disk Staging Table		Memory-Optimized Staging Table	
	Small Row	**Large Row**	**Small Row**	**Large Row**
Staging Table Insert	5,586ms	7,246ms	3,453ms	3,655ms
DimProducts Insert	1,263ms	1,316ms	976ms	993ms
FactSales Insert	13,333ms	13,303ms	13,266ms	13,201ms
Total Time	20,183ms	21,965ms	17,796ms	17,849ms

Table 11-6. *Execution Time of the Tests: With Additional Indexes*

	On-Disk Staging Table		Memory-Optimized Staging Table	
	Small Row	**Large Row**	**Small Row**	**Large Row**
Staging Table Insert	9,233ms	11,656ms	4,751ms	4,893ms
DimProducts Insert	513ms	520ms	506ms	513ms
FactSales Insert	13,163ms	13,276ms	12,283ms	12,300ms
Total Time	22,909ms	25,453ms	17,540ms	17,706ms

As you can see, memory-optimized table `INSERT` performance can be significantly better compared to the on-disk table. The performance gain increases with the row size and when extra indexes are added to the table. Even though extra indexes slow down the insert in both cases, their impact is smaller in the case of memory-optimized tables.

On the other hand, the performance difference during the scans is insignificant. In both cases, the most work is done by accessing `DimProducts` and inserting data into the `FactSales` on-disk tables.

Listing 11-19 illustrates the code that allows us to compare `UPDATE` performance of the tables. The first statement changes a fixed-length column and does not increase the row size. The second statement, on the other hand, increases the size of the row, which triggers the large number of page splits in the on-disk table.

Listing 11-19. ETL Performance: UPDATE Performance

```
update dw.FactSalesETLDisk set Quantity += 1;
update dw.FactSalesETLDisk set OrderNum += '1234567890';

update dw.FactSalesETLMem set Quantity += 1;
update dw.FactSalesETLMem set OrderNum += '1234567890';
```

Tables 11-7 and 11-8 illustrate the average execution time of the tests in my environment. As you can see, the page split operation can significantly degrade update performance for on-disk tables. This is not the case with memory-optimized tables, where new row versions are generated all the time.

Table 11-7. *Execution Time of Update Statements: No Additional Indexes*

	On-Disk Staging Table		Memory-Optimized Staging Table	
	Small Row	**Large Row**	**Small Row**	**Large Row**
Fixed-Length Column Update	2,625ms	2,712ms	2,900ms	2,907ms
Row Size Increase	4,510ms	8,391ms	2,950ms	3,050ms

Table 11-8. *Execution Time of Update Statements: With Additional Indexes*

	On-Disk Staging Table		Memory-Optimized Staging Table	
	Small Row	**Large Row**	**Small Row**	**Large Row**
Fixed-Length Column Update	2,694ms	2,709ms	4,680ms	5,083ms
Row Size Increase	4,456ms	8,561ms	4,756ms	5,186ms

Nonclustered indexes, on the other hand, do not affect update performance of on-disk tables as long as their key columns were not updated. It is not the case with memory-optimized tables where multiple index chains need to be maintained.

As you can see, using memory-optimized tables with a Data Warehouse workload completely fits into the"It depends" category. In some cases, you will benefit from it, while in others performance is degraded. You should carefully test your scenarios before deciding if memory-optimized objects should be used.

Finally, it is worth mentioning that all tests in that section were executed with warm cache and serial execution plans. Physical I/O and parallelism could significantly affect the picture. Moreover, you will get different results if you don't need to persist the staging data and can use temporary and non-durable memory-optimized tables during the processes.

Using In-Memory OLTP as Session - or Object State-Store

Modern software systems have become extremely complex. They consist of a large number of components and services responsible for various tasks, such as interaction with users, data processing, integration with other systems, reporting, and quite a few others. Moreover, modern systems must be scalable and redundant. They need to be able to handle load growth and survive hardware failures and crashes.

The common approach to solving scalability and redundancy issues is to design the systems in a way that permits to deploy and run multiple instances of individual services. It allows adding more servers and instances as the load grows and helps you survive hardware failures by distributing the load across other active servers. The services are usually implemented in stateless way, and they don't store or rely on any local data.

Most systems, however, have data that needs to be shared across the instances. For example, front-end web servers usually need to maintain web session states. Back-end processing services often need to have shared cache with some data.

Historically, there were two approaches to address this issue. The first one was to use dedicated storage/cache and host it somewhere in the system. Remember the old ASP.Net model that used either a SQL Server database or a separate web server to store session data? The problem with this approach is limited scalability and redundancy. Storing session data in web server memory is fast but it is not redundant. A SQL Server database, on the other hand, can be protected but it does not scale well under the load due to page latch contention and other issues.

Another approach was to replicate content of the cache across multiple servers. Each instance worked with the local copy of the cache while another background process distributed the changes to the other servers. Several solutions on the market provide such capability; however, they are usually expensive. In some cases, the license cost for such software could be in the same order of magnitude as SQL Server licenses.

Fortunately, you can use In-Memory OLTP as the solution. In the nutshell, it looks similar to the ASP.Net SQL Server session-store model; however, In-Memory OLTP throughput and performance improvements address the scalability issues of the old on-disk solution.

You can improve performance even further by using non-durable memory-optimized tables. Even though the data will be lost in case of failover, this is acceptable in most cases.

However, the 8,060-byte maximum row size limit introduces challenges to the implementation. It is entirely possible that a serialized object will exceed 8,060 bytes. You can address this by splitting the data into multiple chunks and storing them in multiple rows in memory-optimized table.

You saw an example of a T-SQL implementation earlier in the chapter. However, using T-SQL code and an interop engine will significantly decrease the throughput of the solution. It is better to manage serialization and split/merge functional on the client side.

Listing 11-20 shows the table and natively compiled stored procedures that you can use to store and manipulate the data in the database. The client application calls the `LoadObjectFromStore` and `SaveObjectToStore` stored procedures to load and save the data. The `PurgeExpiredObjects` stored procedure removes expired rows from the table, and it can be called from a SQL Agent or other processes based on the schedule.

Listing 11-19. Implementing Session Store: Database Schema

```
create table dbo.ObjStore
(
    ObjectKey uniqueidentifier not null,
    ExpirationTime datetime2(2) not null,
    ChunkNum smallint not null,
    Data varbinary(8000) not null,

    constraint PK_ObjStore
    primary key nonclustered hash(ObjectKey, ChunkNum)
    with (bucket_count = 131072),

    index IDX_ObjectKey
    nonclustered hash(ObjectKey)
    with (bucket_count = 131072)
)
```

```
with (memory_optimized = on, durability = schema_only);
go

create type dbo.tvpObjData as table
(
    ChunkNum smallint not null
        primary key nonclustered hash
        with (bucket_count = 1024),
    Data varbinary(8000) not null
)
with(memory_optimized=on)
go

create proc dbo.SaveObjectToStore
(
    @ObjectKey uniqueidentifier not null
    ,@ExpirationTime datetime2(2) not null
    ,@ObjData dbo.tvpObjData not null readonly
)
with native_compilation, schemabinding, exec as owner
as
begin atomic
with
(
    transaction isolation level = snapshot
    ,language = N'English'
)
    delete dbo.ObjStore
    where ObjectKey = @ObjectKey

    insert into dbo.ObjStore(ObjectKey, ExpirationTime, ChunkNum, Data)
        select @ObjectKey, @ExpirationTime, ChunkNum, Data
        from @ObjData
end
go

create proc dbo.LoadObjectFromStore
(
    @ObjectKey uniqueidentifier not null
)
with native_compilation, schemabinding, exec as owner
as
begin atomic
with
(
    transaction isolation level = snapshot
    ,language = N'English'
)
```

```
    declare
        @CurrentTime datetime2(2) = sysutcdatetime();

    select t.Data
    from dbo.ObjStore t
    where t.ObjectKey = @ObjectKey and ExpirationTime >= @CurrentTime
    order by t.ChunkNum
end
go

create proc dbo.PurgeExpiredObjects
with native_compilation, schemabinding, exec as owner
as
begin atomic
with
(
    transaction isolation level = snapshot
    ,language = N'English'
)
    declare @CurrentTime
        datetime2(2) = sysutcdatetime();

    delete dbo.ObjStore
    where ExpirationTime < @CurrentTime
end
```

The client implementation includes several static classes. The `ObjStoreUtils` class provides four methods to serialize and deserialize objects into the byte arrays, and split and merge those arrays to/from 8,000-byte chunks. You can see the implementation in Listing 11-20.

Listing 11-20. Implementing Session Store: ObjStoreUtils class

```
public static class ObjStoreUtils
{
    /// <summary>
    /// Serialize object of type T to the byte array
    /// </summary>
    public static byte[] Serialize<T>(T obj)
    {
        using (var ms = new MemoryStream())
        {
            var formatter = new BinaryFormatter();
            formatter.Serialize(ms, obj);

            return ms.ToArray();
        }
    }
```

```
    /// <summary>
    /// Deserialize byte array to the object
    /// </summary>
    public static T Deserialize<T>(byte[] data)
    {
        using (var output = new MemoryStream(data))
        {
            var binForm = new BinaryFormatter();
            return (T) binForm.Deserialize(output);
        }
    }

    /// <summary>
    /// Split byte array to the multiple chunks
    /// </summary>
    public static List<byte[]> Split(byte[] data, int chunkSize)
    {
        var result = new List<byte[]>();

        for (int i = 0; i < data.Length; i += chunkSize)
        {
            int currentChunkSize = chunkSize;
            if (i + chunkSize > data.Length)
                currentChunkSize = data.Length - i;

            var buffer = new byte[currentChunkSize];
            Array.Copy(data, i, buffer, 0, currentChunkSize);

            result.Add(buffer);
        }
        return result;
    }

    /// <summary>
    /// Combine multiple chunks into the byte array
    /// </summary>
    public static byte[] Merge(List<byte[]> arrays)
    {
        var rv = new byte[arrays.Sum(a => a.Length)];
        int offset = 0;
        foreach (byte[] array in arrays)
        {
            Buffer.BlockCopy(array, 0, rv, offset, array.Length);
            offset += array.Length;
        }
        return rv;
    }
}
```

The ObjStoreDataAccess class shown in Listing 11-21 loads and saves binary data to and from the database. It utilizes another static class called DBConnManager, which returns the SqlConnection object to the target database. This class is not shown in the listing.

Listing 11-21. Implementing Session Store: ObjStoreDataAccess class

```
public static class ObjStoreDataAccess
{
    /// <summary>
    /// Saves data to the database
    /// </summary>
    public static void SaveObjectData(Guid key,
                DateTime expirationTime, List<byte[]> chunks)
    {
        using (var cnn = DBConnManager.GetConnection())
        {
            using (var cmd = cnn.CreateCommand())
            {
                cmd.CommandText = "dbo.SaveObjectToStore";
                cmd.CommandType = CommandType.StoredProcedure;
                cmd.Parameters.Add("@ObjectKey",
                    SqlDbType.UniqueIdentifier).Value = key;
                cmd.Parameters.Add("@ExpirationTime",
                    SqlDbType.DateTime2).Value = expirationTime;

                var tvp = new DataTable();
                tvp.Columns.Add("ChunkNum", typeof(short));
                tvp.Columns.Add("ChunkData", typeof(byte[]));

                for(int i=0; i<chunks.Count; i++)
                    tvp.Rows.Add(i, chunks[i]);

                var tvpParam = new SqlParameter("@ObjData",
                     SqlDbType.Structured)
                {
                    TypeName = "dbo.tvpObjData",
                    Value = tvp
                };

                cmd.Parameters.Add(tvpParam);
                cmd.ExecuteNonQuery();
            }
        }
    }
```

```
    /// <summary>
    /// Load data from the database
    /// </summary>
    public List<byte[]> LoadObjectData(Guid key)
    {
        using (var cnn = DBConnManager.GetConnection())
        {
            using (var cmd = cnn.CreateCommand())
            {
                cmd.CommandText = "dbo.LoadObjectFromStore";
                cmd.CommandType = CommandType.StoredProcedure;
                cmd.Parameters.Add("ObjectKey",
                    SqlDbType.UniqueIdentifier).Value = key;

                var result = new List<byte[]>();
                using (var reader = cmd.ExecuteReader())
                {
                    while (reader.Read())
                        result.Add((byte[])reader["Data"]);
                }
                return result;
            }
        }
    }
}
```

Finally, the `ObjStoreService` class shown in Listing 11-22 puts everything together and manages the entire process. It implements two simple methods, `Load` and `Save`, calling the helper classes defined above.

Listing 11-22. Implementing Session Store: ObjStoreService class

```
public static class ObjStoreService
{
    private const int MaxChunkSize = 8000;

    /// <summary>
    /// Saves object in the object store
    /// </summary>
    public static void Save(Guid key,
                DateTime expirationTime, object obj)
    {
        var objectBytes = ObjStoreUtils.Serialize(obj);
        var chunks = ObjStoreUtils.Split(objectBytes, MaxChunkSize);

        ObjStoreDataAccess.SaveObjectData(key, expirationTime, chunks);
    }
```

```
    /// <summary>
    /// Loads object from the object store
    /// </summary>
    public static T Load<T>(Guid key) where T: class
    {
        var chunks = ObjStoreDataAccess.LoadObjectData(key);
        if (chunks.Count == 0)
            return null;
        var objectBytes = ObjStoreUtils.Merge(chunks);

        return ObjStoreUtils.Deserialize<T>(objectBytes);
    }
}
```

Obviously, this is oversimplified example, and production implementation could be significantly more complex, especially if there is the possibility that multiple sessions can update the same object simultaneously. You can implement retry logic or create some sort of object locking management in the system if this is the case.

It is also worth mentioning that you can compress binary data before saving it into the database. The compression will introduce unnecessary overhead in the case of small objects; however, it could provide significant space savings and performance improvements if the objects are large.

I did not include compression code in the example, although you can easily implement it with the `GZipStream` or `DeflateStream` classes.

■ **Note** The code and test application are included in companion materials of this book.

Using In-Memory OLTP in Systems with Mixed Workloads

In-Memory OLTP can provide significant performance improvements in OLTP systems. However, with a Data Warehouse workload, results may vary. The complex queries that perform large scans and aggregations do not necessarily benefit from In-Memory OLTP.

In-Memory OLTP is targeted to the Enterprise market and strong SQL Server teams. It is common to see separate Data Warehouse solutions in those environments. Nevertheless, even in those environments, some degree of reporting and analysis workload is always present in OLTP systems.

The situation is even worse when systems do not have dedicated Data Warehouse and Analysis databases, and OLTP and Data Warehouse queries run against the same data. Moving the data into memory could negatively impact the performance of reporting queries.

One of the solutions in this scenario is to partition the data between memory-optimized and on-disk tables. You can put recent and hot data into memory-optimized tables, keeping old, historical data on-disk. Moreover, it is very common to see different access patterns in the systems when hot data is mainly customer-facing and accessed by OLTP queries while old, historical data is used for reporting and analysis.

Data partitioning also allows you to create a different set of indexes in the tables based on their access patterns. In some cases, you can even use columnstore indexes with the old data, which significantly reduces the storage size and improves the performance of Data Warehouse queries. Finally, you can use partitioned views to hide partitioning details from the client applications.

Listing 11-23 shows an example of such implementation. The memory-optimized table called `RecentOrders` stores the most recent orders that were submitted in 2015. The on-disk `LastYearOrders` table stores the data for 2014. Lastly, the `OldOrders` table stores the old orders that were submitted prior to 2014. The view `Orders` combines the data from all three tables.

Listing 11-23. Data Partitioning: Tables and Views

```
-- Storing Orders with OrderDate >= 2015-01-01
create table dbo.RecentOrders
(
    OrderId int not null identity(1,1),
    OrderDate datetime2(0) not null,
    OrderNum varchar(32)
        collate Latin1_General_100_BIN2 not null,
    CustomerId int not null,
    Amount money not null,
    /* Other columns */
    constraint PK_RecentOrders
    primary key nonclustered hash(OrderId)
    with (bucket_count=1048576),

    index IDX_RecentOrders_CustomerId
    nonclustered(CustomerId)
)
with (memory_optimized=on, durability=schema_and_data)
go

create partition function pfLastYearOrders(datetime2(0))
as range right for values
('2014-04-01','2014-07-01','2014-10-01','2015-01-01')
go

create partition scheme psLastYearOrders
as partition pfLastYearOrders
all to ([LastYearOrders])
go
```

```
create table dbo.LastYearOrders
(
    OrderId int not null,
    OrderDate datetime2(0) not null,
    OrderNum varchar(32)
        collate Latin1_General_100_BIN2 not null,
    CustomerId int not null,
    Amount money not null,
    /* Other columns */
    -- We have to include OrderDate to PK
    -- due to partitioning
    constraint PK_LastYearOrders
    primary key clustered(OrderDate,OrderId)
    with (data_compression=row)
    on psLastYearOrders(OrderDate),

    constraint CHK_LastYearOrders
    check
    (
        OrderDate >= '2014-01-01' and
        OrderDate < '2015-01-01'
    )
);

create nonclustered index IDX_LastYearOrders_CustomerId
on dbo.LastYearOrders(CustomerID)
with (data_compression=row)
on psLastYearOrders(OrderDate);
go

create partition function pfOldOrders(datetime2(0))
as range right for values
(  /* Old intervals */
  '2012-10-01','2013-01-01','2013-04-01'
  ,'2013-07-01','2013-10-01','2014-01-01'
)
go

create partition scheme psOldOrders
as partition pfOldOrders
all to ([OldOrders])
go

create table dbo.OldOrders
(
    OrderId int not null,
    OrderDate datetime2(0) not null,
    OrderNum varchar(32)
        collate Latin1_General_100_BIN2 not null,
    CustomerId int not null,
```

```
    Amount money not null,
    /* Other columns */
    constraint CHK_OldOrders
    check(OrderDate < '2014-01-01')
)
on psOldOrders(OrderDate);

create clustered columnstore index CCI_OldOrders
on dbo.OldOrders
with (data_compression=columnstore_Archive)
on psOldOrders(OrderDate);
go

create view dbo.Orders(OrderId,OrderDate,
    OrderNum,CustomerId,Amount)
as
    select OrderId,OrderDate,OrderNum,CustomerId,Amount
    from dbo.RecentOrders
    where OrderDate >= '2015-01-01'

    union all

    select OrderId,OrderDate,OrderNum,CustomerId,Amount
    from dbo.LastYearOrders

    union all

    select OrderId,OrderDate,OrderNum,CustomerId,Amount
    from dbo.OldOrders
go
```

As you know, memory-optimized tables do not support CHECK constraints, which prevent Query Optimizer from analyzing what data is stored in the RecentOrders table. You can specify that in a where clause of the first SELECT in the view. This will allow SQL Server to eliminate access to the table if queries do not need data from there. You can see this by running the code from Listing 11-24.

Listing 11-24. Data Partitioning: Querying Data

```
select top 10
    CustomerId
    ,sum(Amount) as [TotalSales]
from dbo.Orders
where
    OrderDate >='2013-07-01' and
    OrderDate < '2014-07-01'
group by
    CustomerId
order by
    sum(Amount) desc
```

Figure 11-3 shows the partial execution plan of the query. As you can see, the query does not access the memory-optimized table at all.

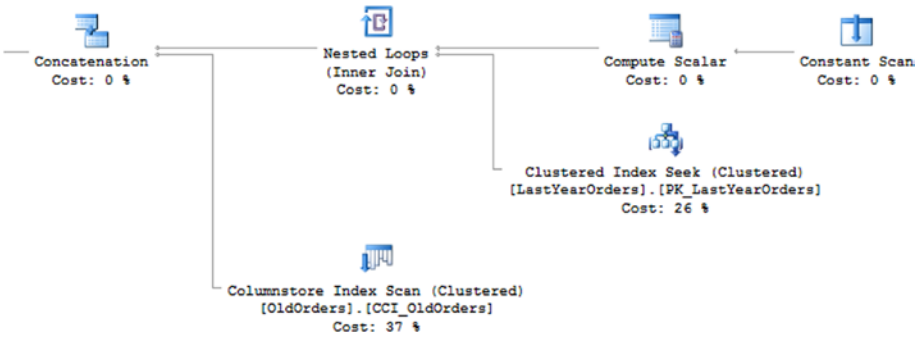


Figure 11-3. *Execution plan of the query*

The biggest downside of this approach is the inability to seam lessly move the data from a memory-optimized table to an on-disk table as the operational period changes. With on-disk tables, it is possible to make the data movement transparent by utilizing the online index rebuild and partition switches. However, it will not work with memory-optimized tables where you have to copy the data to the new location and delete it from the source table afterwards.

This should not be a problem if the system has a maintenance window when such operations can be performed. Otherwise, you will need to put significant development efforts into preventing customers from modifying data on the move.

■ **Note** Chapter 15 in my book *Pro SQL Server Internals* discusses various data partitioning aspects including how to move data between different tables and file groups while keeping it transparent to the users.

Summary

In-Memory OLTP can dramatically improve the performance of OLTP systems. However, it can lead to large implementation cost especially when you need to migrate existing systems. You should perform a cost/benefits analysis, making sure that the implementation cost is acceptable. It is still possible to benefit from In-Memory OLTP objects even when you cannot utilize the technology in its full scope.

Some of the In-Memory OLTP limitations can be addressed in the code. You can split the data between multiple tables to work around the 8,060-byte maximum row size limitation or, alternatively, store large objects in multiple rows in the table. Uniqueness and referential integrity can be enforced with `REPEATABLE READ` and `SERIALIZABLE` transaction isolation levels.

You should be careful when using In-Memory OLTP with a Data Warehouse workload and queries that perform large scans. While it can help in some scenarios, it could degrade performance of the systems in others. You can implement data partitioning, combining the data from memory-optimized and on-disk tables when this is the case.

APPENDIX A

Memory Pointers Management

This chapter explains how SQL Server works with memory pointers that link In-Memory OLTP objects together.

Memory Pointers Management

The In-Memory OLTP Engine relies on memory pointers, using them to link objects together. For example, pointers embedded into data rows link them into the index chains, which, in turn, are referenced by the hash and nonclustered index objects. The lock- and latch-free nature of In-Memory OLTP adds the challenge of managing memory pointers in highly volatile environments where multiple sessions can try to simultaneously change them, overwriting each other's changes.

Consider the situation when multiple sessions are trying to insert rows into the same index row chain. Each session traverses that chain to locate the last row there and update its pointer with the address of the newly created row. SQL Server must guarantee that every row is added to the chain even when there are multiple sessions running in different parallel threads and they are trying to perform that pointer update simultaneously.

SQL Server uses an `InterlockedCompareExchange` mechanism to guarantee that. `InterlockedCompareExchange` functions change the value of the pointer, checking that the existing (*pre-update*) value matches the expected (*old*) value provided as another parameter. Only when the check succeeds is the pointer value updated.

To illustrate this, assume that you have two sessions that want to simultaneously insert new delta records for the same nonclustered index leaf page. As a first step, shown in Figure A-1, the sessions create delta records and set their pointers to a page based on the address from the mapping table.

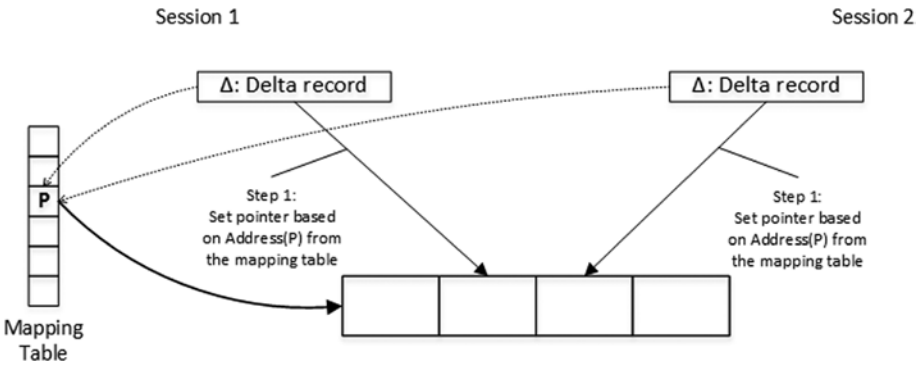


Figure A-1. *Data modifications and concurrency: Step 1*

In the next step, both sessions call the `InterlockedCompareExchange` function to try to update the mapping table by changing the reference from a page to the delta records they just created. `InterlockedCompareExchange` serializes the update of the mapping table element and changes it only if its current preupdate value matches the old pointer (address of the page) provided as the parameter. The first `InterlockedCompareExchange` call succeeds. The second call, however, fails because the mapping table element references the delta record from another session rather than the page. Therefore, the second session needs to redo or rollback the action based on the requirements and a use case.

Figure A-2 illustrates such a scenario. As you can see, with the exception of a very short serialization during the `InterlockedCompareExchange` call, there is no locking or latching of the data during the modifications.

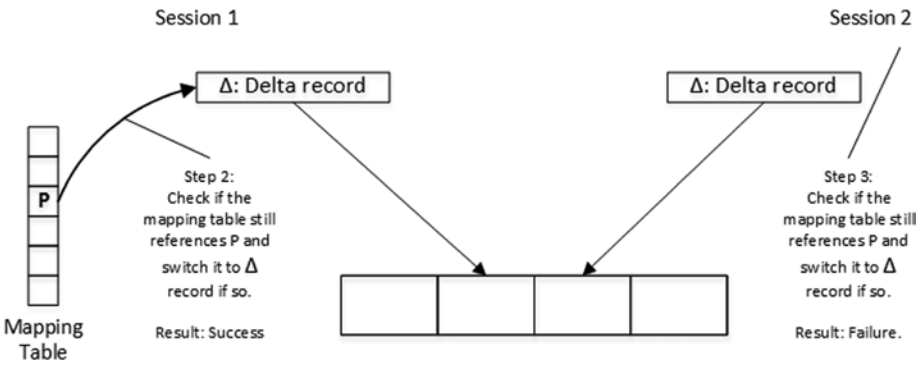


Figure A-2. *Data modifications and concurrency: Step 2*

SQL Server uses the same approach with `InterlockedCompareExchange` every time the pointer chain needs to be preserved, such as when it creates another version of a row during an update, when it needs to change a pointer in the index mapping or hash tables, and in quite a few other cases.

Summary

SQL Server uses an `InterlockedCompareExchange` mechanism to guarantee that multiple sessions cannot update the same memory pointers simultaneously, losing references to each other's objects. `InterlockedCompareExchange` functions change the value of the pointer, checking that the existing (*preupdate*) value matches the expected (*old*) value provided as another parameter. Only when the check succeeds is the pointer value updated.

APPENDIX B

Page Splitting and Page Merging in Nonclustered Indexes

This appendix provides an overview of nonclustered index internal operations, such as page splitting and page merging.

Nonclustered Indexes Internal Maintenance

The In-Memory OLTP Engine has several internal operations to maintain the structure of nonclustered indexes. As you already know from Chapter 5, *page consolidation* rebuilds the nonclustered index page, consolidating all changes defined by the page delta records. It helps avoid the performance hit introduced by long delta record chains. The newly created page has the same PID in the mapping table and replaces the old page, which is marked for garbage collection.

Two other processes can create new index pages, *page splitting* and *page merging*. Both are complex actions and deserve detailed explanations of their internal implementation.

Page Splitting

Page splitting occurs when a page does not have enough free space to accommodate a new data row. Even though the process is similar to a B-Tree on-disk index page split, there is one conceptual difference. In B-Tree indexes, the page split moves the part of the data to the new data page, freeing up space on the original page. In Bw-Tree indexes, however, the pages are non-modifiable, and SQL Server replaces the old page with two new ones, splitting the data between them.

Let's look at this situation in more detail. Figure B-1 shows the internal and leaf pages of the nonclustered index. Let's assume that one of the sessions wants to insert a row with a key of value *Bob*.

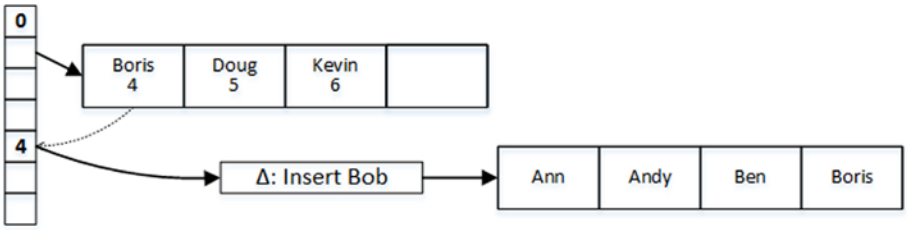


Figure B-1. *Page splitting: Initial state*

When the delta record is created, SQL Server adjusts the delta records statistics on the index page and detects that there is no space on the page to accommodate the new index value once the delta records are consolidated. It triggers a page split process, which is done in two atomic steps.

In the first step, SQL Server creates two new leaf-level pages and splits the old page values between them. After that, it repoints the mapping table to the first newly created page and marks the old page and the delta records for garbage collection; Figure B-2 illustrates this state. At this state, there are no references to the second newly created leaf-level page from the internal pages. The first leaf-level page, however, maintains the link between pages (through the mapping table), and SQL Server is able to access and scan the second page if needed.

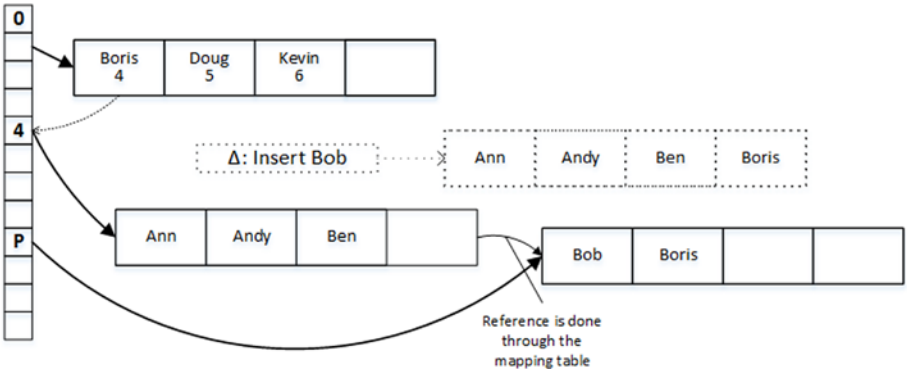


Figure B-2. *Page splitting: First step*

During the second step, SQL Server creates another internal page with key values that represent the new leaf-level page layout. When the new page is created, SQL Server switches the pointer in the mapping table and marks the old internal page for garbage collection. Figure B-3 illustrates this action.

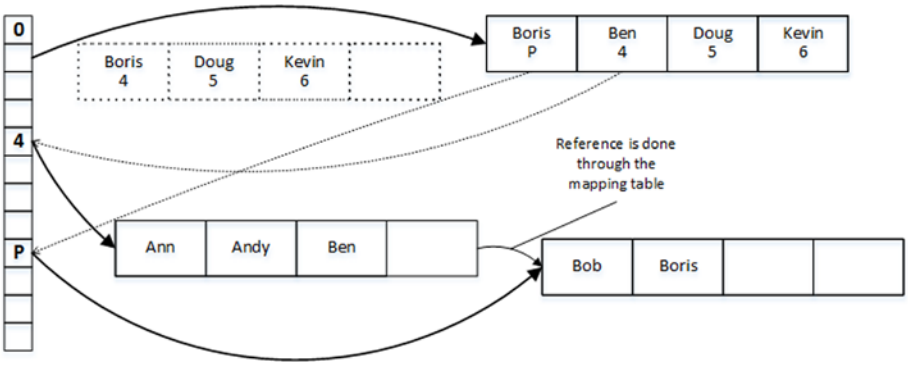


Figure B-3. *Page splitting: Second step*

Eventually, the old data pages and delta records are deallocated by the garbage collection process.

Page Merging

Page merging occurs when a delete operation leaves an index page less than 10% from the maximum page size, which is 8KB now, or when an index page contains just a single row. During this operation, SQL Server merges the data from two adjacent index pages, replacing them with the new, combined, data page.

Assume that you have the page layout shown in Figure B-3, and you want to delete the index key value *Bob*, which means that all data rows with the name *Bob* have been already deleted. This leaves an index page with the single value *Boris*, which triggers page merging.

In the first step, SQL Server creates a delete delta record for Bob and another special kind of delta record called *merge delta*. Figure B-4 illustrates the layout after the first step.

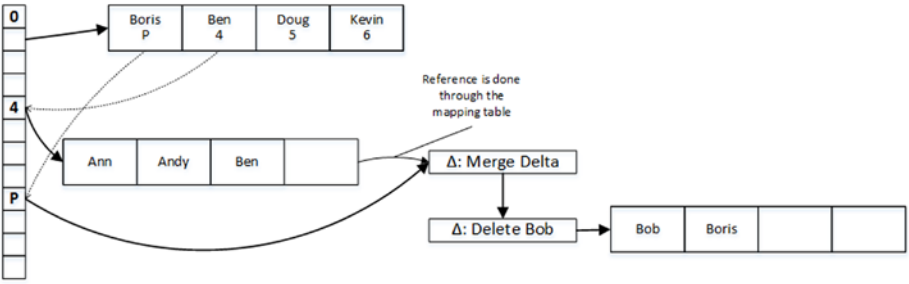


Figure B-4. *Page merging: First step*

During the second step of page merging, SQL Server creates a new internal page that does not reference the leaf-level page that it is about to be merged. After that, SQL Server switches the mapping table to point to the newly created internal page and marks the old page for garbage collection. Figure B-5 illustrates this action.

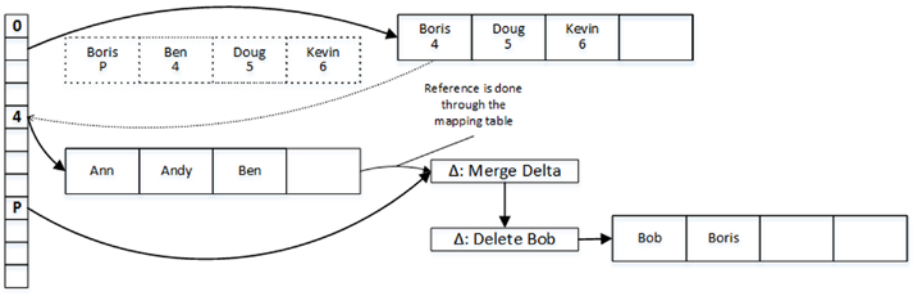


Figure B-5. *Page merging: Second step*

Finally, SQL Server builds a new leaf-level page, copying the *Boris* value there. After the new page is created, it updates the mapping table and marks the old pages and delta records for garbage collection.

Figure B-6 shows the final data layout after page merging is completed.

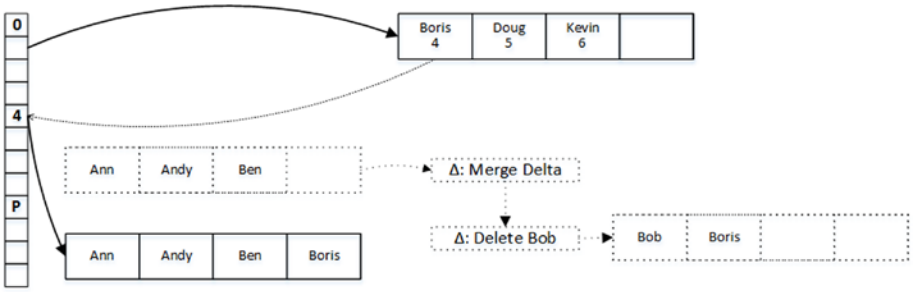


Figure B-6. *Page merging: Third (final) step*

You can get page consolidation, merging, and splitting statistics from the `sys.dm_db_xtp_nonclustered_index_stats` view.

■ **Note** You can read documentation about the `sys.dm_db_xtp_nonclustered_index_stats` view at `https://msdn.microsoft.com/en-us/library/dn645468.aspx`.

Summary

The In-Memory OLTP Engine uses several internal operations to maintain the structure of nonclustered indexes. Page consolidation rebuilds the index page, combining page data with the delta records. It helps avoid the performance impact introduced by long delta records chains.

Page splitting occurs when the index page does not have enough space to accommodate the new rows. In contrast to page splitting on-disk B-Tree indexes, which moves part of the data to the new page, Bw-Tree page splitting replaces the old data page with new pages that contain the data.

Page merging occurs when an index page is less than 10% of the maximum page size or when it has just a single row. SQL Server merges the data from adjacent data pages and replaces them with the new page with the merged data.

APPENDIX C

Analyzing the States of Checkpoint File Pairs

SQL Server persists data from durable memory-optimized tables in checkpoint file pairs. This appendix demonstrates how to analyze their states using the `sys.db_dm_xtp_checkpoint_files` view and shows the state transitions through the CFP lifetime.

Sys.db_dm_xtp_checkpoint_files View

The `sys.db_dm_xtp_checkpoint_files` view provides information about database checkpoint files, including their state, size, and physical location. We will use this view extensively in this appendix. Let's look at the most important columns.

- The `container_id` and `container_guid` columns provide information about the `FILESTREAM` container to which the checkpoint file belongs. `Container_id` corresponds to the `file_id` column in the `sys.database_files` view.
- `checkpoint_file_id` is a GUID that represents the ID of the file.
- `checkpoint_pair_file_id` is the ID of the second, data or delta, file in the pair.
- `relative_file_path` shows the relative file path in the container.
- `state` and `state_desc` describe the state of the file. As you already know from Chapter 8, the checkpoint file pair can be in one of the following states (the number represents the `state` column value): `(0) - PRECREATED`, `(1) - UNDER CONSTRUCTION`, `(2) - ACTIVE`, `(3) - MERGE TARGET`, `(4) - MERGED SOURCE`, `(5) - REQUIRED FOR BACKUP/HA`, `(6) - IN TRANSITION TO TOMBSTONE`, `(7) - TOMBSTONE`.
- `file_type` and `file_type_desc` describe the type of file: `(0) - DATA_FILE`, `(1) - DELTA_FILE`. These columns return `NULL` if the CFP is in the `TOMBSTONE` state.

- `lower_bound_tsn` and `upper_bound_tsn` indicate the timestamp of the earliest and latest transactions covered by the data file. These columns are populated only for `ACTIVE`, `MERGE TARGET`, and `MERGED SOURCE` states.
- `internal_storage_slot` is the index of the file in the internal storage array. As you already know from Chapter 8, In-Memory OLTP persists the metadata information about checkpoint file pairs in an internal 8,192-slot array. This column of the view is populated only for `UNDER CONSTRUCTION` and `ACTIVE` CFPs.
- `file_size_in_bytes` and `file_size_used_in_bytes` provide information about file size and space used in the file. When the file is still being populated, `file_size_used_in_bytes` is updated at the time of the checkpoint event. These columns return `NULL` for files in the `REQUIRED FOR BACKUP/HA`, `IN TRANSITION TO TOMBSTONE`, and `TOMBSTONE` states.
- `inserted_row_count`and `deleted_row_count` provide the number of rows in the data and delta files. `Drop_table_deleted_row_count` shows the number of rows in the tables that were dropped.

Let's use this view and analyze the state transitions of the checkpoint file pairs.

The Lifetime of Checkpoint File Pairs

As the first step in this test, let's enable the undocumented trace flag `TF9851` using the `DBCC TRACEON(9851,-1)` command. This trace flag disables the automatic merge process, which will allow you to have more control over your test environment.

■ **Important** Do not set `TF9851` in production.

Let's create the database with an In-Memory OLTP file group and perform the full backup starting the backup chain, as shown in Listing C-1. I am doing it in the test environment and not following best practices (such as placing In-Memory OLTP and on-disk data on different drives, creating secondary file groups for on-disk data, and a few others). Obviously, you should remember to follow best practices when you design your real databases.

Listing C-1. Creating a Database and Performing Backup

```
create database [AppendixC]
on primary
(
    name = N'AppendixC'
    ,filename = N'C:\Data\AppendixC.mdf'
),
filegroup HKData CONTAINS MEMORY_OPTIMIZED_DATA
(
    name = N'AppendixC_HKData'
    ,filename = N'C:\Data\HKData\AppendixC'
)
log on
(
    name = N'AppendixC_Log'
    ,filename = N'C:\Data\AppendixC_log.ldf'
)
go

backup database [AppendixC]
to disk = N'C:\Data\Backups\AppendixC.bak'
with noformat, init, name = 'AppendixC - Full', compression;
```

The database is currently empty and, therefore, it does not have any checkpoint file pairs created. You can confirm this by querying the `sys.dm_db_xtp_checkpoint_files` view, as shown in Listing C-2.

Listing C-2. Checking Checkpoint File Pairs

```
use [AppendixC]
go

select
    checkpoint_file_id
    ,checkpoint_pair_file_id
    ,file_type_desc
    ,state_desc
    ,file_size_in_bytes
    ,relative_file_path
from
    sys.dm_db_xtp_checkpoint_files
order by
    state, file_type
```

Figure C-1 shows that result set is empty and that the `sys.dm_db_xtp_checkpoint_files` view does not return any data.

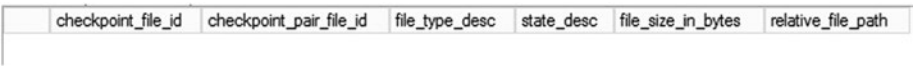


Figure C-1. *State of checkpoint file pairs after database creation*

As the next step, let's create a durable memory-optimized table, as shown in Listing C-3.

Listing C-3. Creating a Durable Memory-Optimized Table

```
create table dbo.HKData
(
    ID int not null,
    Placeholder char(8000) not null,

    constraint PK_HKData
    primary key nonclustered hash(ID)
    with (bucket_count=10000),
)
with
(
    memory_optimized=on
    ,durability=schema_and_data
)
```

If you check the state of the checkpoint file pairs now and run the code from Listing C-2 again, you will see the output shown in Figure C-2. The total number of files and their size may be different in your environment and will depend on the hardware. My test virtual machine has four vCPU and 8GB of RAM, so I have eight checkpoint file pairs in the `PRECREATED` state with 16MB data and 1MB delta files. I also have one CFP in the `UNDER CONSTRUCTION` state.

	checkpoint_fil...	checkpoint_pai...	file_type_desc	state_desc	file_size_in_bytes	relative_file_path
1	24E75416-83...	10DA62F5-F5...	DATA	PRECREATED	16777216	c7e7c31a-fa17-4b
2	71687FCD-F0...	C27CA2A6-97...	DATA	PRECREATED	16777216	c7e7c31a-fa17-4b
3	9FDE498B-E6...	A325100C-F90...	DATA	PRECREATED	16777216	c7e7c31a-fa17-4b
4	6CD3E025-58...	8AF404A7-72C...	DATA	PRECREATED	16777216	c7e7c31a-fa17-4b
5	CAC4D922-E...	BC9FF414-F67...	DATA	PRECREATED	16777216	c7e7c31a-fa17-4b
6	5A8C6B48-33...	862A5885-783...	DATA	PRECREATED	16777216	c7e7c31a-fa17-4b
7	DC6F02FB-19...	B47A4BE8-42...	DATA	PRECREATED	16777216	c7e7c31a-fa17-4b
8	D63FD088-25...	73A2E8AA-66...	DATA	PRECREATED	16777216	c7e7c31a-fa17-4b
9	C27CA2A6-97...	71687FCD-F0F...	DELTA	PRECREATED	1048576	c7e7c31a-fa17-4b
10	10DA62F5-F5...	24E75416-83B...	DELTA	PRECREATED	1048576	c7e7c31a-fa17-4b
11	B47A4BE8-42...	DC6F02FB-19...	DELTA	PRECREATED	1048576	c7e7c31a-fa17-4b
12	73A2E8AA-66...	D63FD088-25...	DELTA	PRECREATED	1048576	c7e7c31a-fa17-4b
13	BC9FF414-F6...	CAC4D922-E9...	DELTA	PRECREATED	1048576	c7e7c31a-fa17-4b
14	A325100C-F9...	9FDE498B-E6...	DELTA	PRECREATED	1048576	c7e7c31a-fa17-4b
15	8AF404A7-72...	6CD3E025-58...	DELTA	PRECREATED	1048576	c7e7c31a-fa17-4b
16	862A5885-78...	5A8C6B48-33...	DELTA	PRECREATED	1048576	c7e7c31a-fa17-4b
17	0D073B1E-6...	DE25838A-DE...	DATA	UNDER CONSTRUCTION	16777216	c7e7c31a-fa17-4b
18	DE25838A-D...	0D073B1E-6D...	DELTA	UNDER CONSTRUCTION	1048576	c7e7c31a-fa17-4b

Figure C-2. *State of checkpoint file pairs after creating the durable memory-optimized table*

Let's enlarge the output for the files from the `UNDER CONSTRUCTION` CFP, as shown in Figure C-3. As you can see, the `checkpoint_pair_file_id` values reference the `checkpoint_file_id` of the second file in the pair.

	checkpoint_file_id	checkpoint_pair_file_id	file_type_desc	state_desc
1	0D073B1E-6D16-4E82-B7B0-A3AE05AA39D2	DE25838A-DE71-4FFE-9CC1-6EE4C3E599E1	DATA	UNDER CONSTRUCTION
2	DE25838A-DE71-4FFE-9CC1-6EE4C3E599E1	0D073B1E-6D16-4E82-B7B0-A3AE05AA39D2	DELTA	UNDER CONSTRUCTION

	file_size_in_bytes	relative_file_path
1	16777216	c7e7c31a-fa17-4b7d-8abb-fc921e1e8393\c7c0eaa2-4c17-48a8-b3b7-d5b19121f7ac\00000024-00000198-0061
2	1048576	c7e7c31a-fa17-4b7d-8abb-fc921e1e8393\c7c0eaa2-4c17-48a8-b3b7-d5b19121f7ac\00000024-000001c0-0002

Figure C-3. *UNDER CONSTRUCTION checkpoint file pair*

`Relative_file_path` provides the path to the file relative to the `FILESTREAM` container in the In-Memory OLTP file group. Figure C-4 shows the checkpoint files in the folder on the disk.

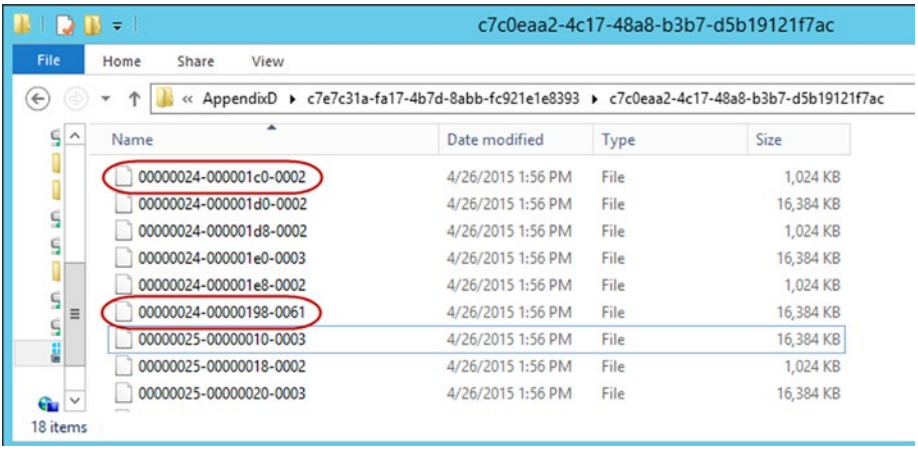


Figure C-4. *Checkpoint files on disk*

Now, populate the `dbo.HKData` table with 1,000 rows and check the status of the checkpoint files, as shown in Listing C-4. The query filters out the checkpoint file pairs in the `PRECREATED` state from the output.

Listing C-4. Populating the dbo.HKData Table and Checking the States of the CFPs

```
;with N1(C) as (select 0 union all select 0) -- 2 rows
,N2(C) as (select 0 from N1 as t1 cross join N1 as t2) -- 4 rows
,N3(C) as (select 0 from N2 as t1 cross join N2 as t2) -- 16 rows
,N4(C) as (select 0 from N3 as t1 cross join N3 as t2) -- 256 rows
,N5(C) as (select 0 from N4 as t1 cross join N4 as t2) -- 65,536 rows
,Ids(Id) as (select row_number() over (order by (select null)) from N5)
insert into dbo.HKData(Id, Placeholder)
    select Id, Replicate('0',8000)
    from ids
    where Id <= 1000;

select
    checkpoint_file_id
    ,file_type_desc
    ,state_desc
    ,lower_bound_tsn
    ,upper_bound_tsn
    ,file_size_in_bytes
    ,inserted_row_count
    ,deleted_row_count
from
    sys.dm_db_xtp_checkpoint_files
where
    state_desc <> 'PRECREATED'
order by
    state, file_type;
```

As you can see in Figure C-5, SQL Server populates the data file in the UNDER CONSTRUCTION CFP by inserting 1,000 rows there. The NULL value in the lower_bound_tsn column indicates that this CFP covers transactions from the time of database creation. Similarly, the NULL value in the upper_bound_tsn column indicates that this CFP covers current transactions.

	checkpoint_file_id	file_type_desc	state_desc
1	0D073B1E-6D16-4E82-B7B0-A3AE05AA39D2	DATA	UNDER CONSTRUCTION
2	DE25838A-DE71-4FFE-9CC1-6EE4C3E599E1	DELTA	UNDER CONSTRUCTION

	lower_bound_tsn	upper_bound_tsn	file_size_in_bytes	inserted_row_count	deleted_row_count
1	NULL	NULL	16777216	1000	NULL
2	NULL	NULL	1048576	NULL	0

Figure C-5. *UNDER CONSTRUCTION CFP state after insert*

Let's run a manual CHECKPOINT and check the status of checkpoint file pairs, as shown in Listing C-5.

Listing C-5. Forcing CHECKPOINT and Checking the Status of CFPs

```
checkpoint
go

select
    checkpoint_file_id
    ,file_type_desc
    ,state_desc
    ,lower_bound_tsn
    ,upper_bound_tsn
    ,file_size_in_bytes
    ,file_size_used_in_bytes
    ,inserted_row_count
    ,deleted_row_count
from
    sys.dm_db_xtp_checkpoint_files
where
    state_desc <> 'PRECREATED'
order by
    state, file_type;
```

As you can see in Figure C-6, the CHECKPOINT operation transitions the UNDER CONSTRUCTION CFP to the ACTIVE state. The upper_bound_tsn columns are now populated, indicating the maximum timestamp for transactions covered by the checkpoint file pair.

	checkpoint_file_id	file_type_desc	state_desc	lower_bound_tsn	upper_bound_tsn
1	0D073B1E-6D16-4E82-B7B0-A3AE05AA39D2	DATA	ACTIVE	NULL	4
2	DE25838A-DE71-4FFE-9CC1-6EE4C3E599E1	DELTA	ACTIVE	NULL	4

	file_size_in_bytes	file_size_used_in_bytes	inserted_row_count	deleted_row_count
1	16777216	8036000	1000	NULL
2	1048576	0	NULL	0

Figure C-6. *The CFP state after CHECKPOINT*

Let's insert another 1,000 rows to the `dbo.HKData` table and check the status of the CFPs. Listing C-6 shows the code to perform this.

Listing C-6. Populating the dbo.HKData Table with Another Batch of Rows and Checking the States of the CFPs Afterwards

```
;with N1(C) as (select 0 union all select 0) -- 2 rows
,N2(C) as (select 0 from N1 as t1 cross join N1 as t2) -- 4 rows
,N3(C) as (select 0 from N2 as t1 cross join N2 as t2) -- 16 rows
,N4(C) as (select 0 from N3 as t1 cross join N3 as t2) -- 256 rows
,N5(C) as (select 0 from N4 as t1 cross join N4 as t2) -- 65,536 rows
,Ids(Id) as (select row_number() over (order by (select null)) from N5)
insert into dbo.HKData(Id, Placeholder)
    select 1000 + Id, Replicate('0',8000)
    from ids
    where Id <= 1000;

select
    checkpoint_file_id
    ,file_type_desc
    ,state_desc
    ,lower_bound_tsn
    ,upper_bound_tsn
    ,file_size_in_bytes
    ,inserted_row_count
    ,deleted_row_count
from
    sys.dm_db_xtp_checkpoint_files
where
    state_desc <> 'PRECREATED'
order by
    state, file_type;
```

Figure C-7 shows the states of the checkpoint file pairs after the second insert. As you can see, SQL Server creates another CFP in the `UNDER CONSTRUCTION` state with `lower_bound_tsn = 4`.

	checkpoint_file_id	file_type_desc	state_desc	lower_bound_tsn	upper_bound_tsn
1	9FDE498B-E62D-471B-A4C9-5CB0875CF9EA	DATA	UNDER CONSTRUCTION	4	NULL
2	A325100C-F90E-479B-B116-2B9FC5055B3E	DELTA	UNDER CONSTRUCTION	4	NULL
3	0D073B1E-6D16-4E82-B7B0-A3AE05AA39D2	DATA	ACTIVE	NULL	4
4	DE25838A-DE71-4FFE-9CC1-6EE4C3E599E1	DELTA	ACTIVE	NULL	4

	file_size_in_bytes	file_size_used_in_bytes	inserted_row_count	deleted_row_count
1	16777216	0	1000	NULL
2	1048576	0	NULL	0
3	16777216	8036000	1000	NULL
4	1048576	0	NULL	0

Figure C-7. *States of CFPsafter the second INSERT*

Another CHECKPOINT would transition the `UNDER CONSTRUCTION` CFP to the `ACTIVE` state, as shown in Figure C-8. You can force it by running the code from Listing C-5 again. At this point, you have two `ACTIVE` checkpoint file pairs covering different ranges of transaction timestamps.

	checkpoint_file_id	file_type_desc	state_desc	lower_bound_tsn	upper_bound_tsn
1	9FDE498B-E62D-471B-A4C9-5CB0875CF9EA	DATA	ACTIVE	4	8
2	0D073B1E-6D16-4E82-B7B0-A3AE05AA39D2	DATA	ACTIVE	NULL	4
3	A325100C-F90E-479B-B116-2B9FC5055B3E	DELTA	ACTIVE	4	8
4	DE25838A-DE71-4FFE-9CC1-6EE4C3E599E1	DELTA	ACTIVE	NULL	4

	file_size_in_bytes	file_size_used_in_bytes	inserted_row_count	deleted_row_count
1	16777216	8036000	1000	NULL
2	16777216	8036000	1000	NULL
3	1048576	0	NULL	0
4	1048576	0	NULL	0

Figure C-8. *States of CFPs after second CHECKPOINT*

As the next step, let's delete 66.7% of the rows from the table, as shown in Listing C-7. In this listing, you are also running the query that combines the information about the data and delta files, and demonstrates that both checkpoint file pairs are mostly empty.

Listing C-7. Deleting 66.7% of the Rows from the Table

```
delete from dbo.HKData
where ID % 3 <> 0;

select
    data.checkpoint_file_id
    ,data.state_desc
    ,data.lower_bound_tsn
    ,data.upper_bound_tsn
    ,data.inserted_row_count
    ,delta.deleted_row_count
```

```
    ,convert(decimal(5,2),
        100. - 100. * delta.deleted_row_count /
            IIF(data.inserted_row_count = 0,1,data.inserted_row_count)
        ) as [% Full]
from
    sys.dm_db_xtp_checkpoint_files data join
        sys.dm_db_xtp_checkpoint_files delta on
            data.checkpoint_pair_file_id =
                delta.checkpoint_file_id
where
    data.file_type_desc = 'DATA' and
    data.state_desc <> 'PRECREATED';
```

As you can see in Figure C-9, both files are just 33.3% full so they are perfect candidates for the merge.

	checkpoint_file_id	state_desc	lower_bound_tsn	upper_bound_tsn	inserted_row_count	deleted_row_count	% Full
1	9FDE498B-E62D-471B-A4C9-5CB0875CF9EA	ACTIVE	4	8	1000	667	33.30
2	0D073B1E-6D16-4E82-B7B0-A3AE05AA39D2	ACTIVE	NULL	4	1000	667	33.30

Figure C-9. *States after deletion*

You can trigger the merge by calling the `sys.sp_xtp_merge_checkpoint_files` system stored procedure. This procedure requires you to provide the lower and upper bounds for the merge and it does not accept `NULL` as the parameter value. You can use any `tsn`, which is covered by the CFP file participating in the merge.

As already discussed, in most cases you can rely on the automatic merge and do not need to call this procedure manually. One of the cases when manual merge is beneficial is the situation when the size of the data in the durable memory-optimized tables is close to 256GB and you want granular control over the merge process, avoiding situations when you do not have enough space in the checkpoint file pairs to store the data. Listing C-8 shows the code that calls the stored procedure.

Listing C-8. Trigerring the Merge Process

```
exec sys.sp_xtp_merge_checkpoint_files
    @database_name = 'AppendixC'
    ,@transaction_lower_bound = 1
    ,@transaction_upper_bound = 8;
```

You can check the status of merge requests by examining the `sys.dm_db_xtp_merge_requests` view, as shown in Listing C-9. Figure C-10 illustrates the output of the query.

Listing C-9. Checking the Status of the Merge Request

```
select
    request_state_desc
    ,destination_file_id
    ,lower_bound_tsn
    ,upper_bound_tsn
    ,source0_file_id
    ,source1_file_id
from
    sys.dm_db_xtp_merge_requests;
```

	request_state_desc	destination_file_id	lower_bound_tsn	upper_bound_tsn
1	PENDING	B99328BF-9FC2-47BC-86D1-708276CEDED1	0	8

	source0_file_id	source1_file_id
1	0D073B1E-6D16-4E82-B7B0-A3AE05AA39D2	9FDE498B-E62D-471B-A4C9-5CB0875CF9EA

Figure C-10. *Merge request status*

■ **Note** You can read more about the `sys.sp_xtp_merge_checkpoint_files` stored procedure at https://msdn.microsoft.com/en-us/library/dn198330.aspx. More information about `sys.dm_db_xtp_merge_requests` is available at https://msdn.microsoft.com/en-us/library/dn465868.aspx.

Figure C-11 illustrates the state of checkpoint file pairs after the merge is initiated. As you can see, SQL Server creates the new checkpoint file pair in the `MERGE TARGET` state and merges data from `ACTIVE` CFPs there. You can also see the correlation between the `checkpoint_file_id` of CFPs with `destination_file_id` and `source_file_id` columns in `sys.dm_db_xtp_merge_requests` view.

	checkpoint_file_id	file_type_desc	state_desc	lower_bound_tsn	upper_bound_tsn
1	9FDE498B-E62D-471B-A4C9-5CB0875CF9EA	DATA	ACTIVE	4	8
2	0D073B1E-6D16-4E82-B7B0-A3AE05AA39D2	DATA	ACTIVE	NULL	4
3	DE25838A-DE71-4FFE-9CC1-6EE4C3E599E1	DELTA	ACTIVE	NULL	4
4	A325100C-F90E-479B-B116-2B9FC5055B3E	DELTA	ACTIVE	4	8
5	B99328BF-9FC2-47BC-86D1-708276CEDED1	DATA	MERGE TARGET	NULL	NULL
6	7B21ABEB-77E3-4775-B3B6-44CE273A713C	DELTA	MERGE TARGET	NULL	NULL

	file_size_in_bytes	file_size_used_in_bytes	inserted_row_count	deleted_row_count
1	16777216	8036000	1000	NULL
2	16777216	8036000	1000	NULL
3	1048576	0	NULL	667
4	1048576	0	NULL	667
5	16777216	0	2000	NULL
6	1048576	0	NULL	0

Figure C-11. *The state of checkpoint file pairs after the merge is initiated*

The next `CHECKPOINT` will transition the checkpoint file pairs that participated in merge from `ACTIVE` to `MERGED SOURCE` and from `MERGE TARGET` CFP to `ACTIVE` states. Figure C-12 demonstrates this. Now the merge is considered to be complete and the request state value from the `sys.dm_db_xtp_merge_requests` view is changed to `INSTALLED`.

	checkpoint_file_id	file_type_desc	state_desc	lower_bound_tsn	upper_bound_tsn
1	B99328BF-9FC2-47BC-86D1-708276CEDED1	DATA	ACTIVE	NULL	8
2	7B21ABEB-77E3-4775-B3B6-44CE273A713C	DELTA	ACTIVE	NULL	8
3	0D073B1E-6D16-4E82-B7B0-A3AE05AA39D2	DATA	MERGED SOURCE	NULL	4
4	9FDE498B-E62D-471B-A4C9-5CB0875CF9EA	DATA	MERGED SOURCE	4	8
5	A325100C-F90E-479B-B116-2B9FC5055B3E	DELTA	MERGED SOURCE	4	8
6	DE25838A-DE71-4FFE-9CC1-6EE4C3E599E1	DELTA	MERGED SOURCE	NULL	4

	file_size_in_bytes	file_size_used_in_bytes	inserted_row_count	deleted_row_count
1	16777216	16072000	2000	NULL
2	1048576	37352	NULL	1334
3	16777216	16072000	1000	NULL
4	16777216	16072000	1000	NULL
5	1048576	18676	NULL	667
6	1048576	18676	NULL	667

Figure C-12. *The state of the checkpoint file pairs after the merge is completed*

After the next `CHECKPOINT`, the `MERGED SOURCE` CFPs will be transitioned to the `REQUIRED FOR BACKUP/HA` state, as shown in Figure C-13.

	checkpoint_file_id	file_type_desc	state_desc	lower_bound_tsn	upper_bound_tsn
1	B99328BF-9FC2-47BC-86D1-708276CEDED1	DATA	ACTIVE	NULL	8
2	7B21ABEB-77E3-4775-B3B6-44CE273A713C	DELTA	ACTIVE	NULL	8
3	9FDE498B-E62D-471B-A4C9-5CB0875CF9EA	DATA	REQUIRED FOR BACKUP/HA	NULL	NULL
4	0D073B1E-6D16-4E82-B7B0-A3AE05AA39D2	DATA	REQUIRED FOR BACKUP/HA	NULL	NULL
5	DE25838A-DE71-4FFE-9CC1-6EE4C3E599E1	DELTA	REQUIRED FOR BACKUP/HA	NULL	NULL
6	A325100C-F90E-479B-B116-2B9FC5055B3E	DELTA	REQUIRED FOR BACKUP/HA	NULL	NULL

	file_size_in_bytes	file_size_used_in_bytes	inserted_row_count	deleted_row_count
1	16777216	16072000	2000	NULL
2	1048576	37352	NULL	1334
3	NULL	NULL	NULL	NULL
4	NULL	NULL	NULL	NULL
5	NULL	NULL	NULL	NULL
6	NULL	NULL	NULL	NULL

Figure C-13. *The MERGED SOURCE CFPs are transitioned to the REQUIRED FOR BACKUP/HA state after the next checkpoint*

After the transaction log backup is taken, log records are transmitted to secondary nodes, and the checkpoint event occurs, these CFPs are eventually picked up by the garbage collection thread and moved to `IN TRANSITION TO TOMBSTONE` and `TOMBSTONE` states and eventually deallocated. You can also force manual garbage collection by calling the `sys.sp_xtp_checkpoint_force_garbage_collection` stored procedure. Listing C-10 illustrates this.

Listing C-10. Performing Log Backup and Forcing Garbage Collection

```
backup log [AppendixC]
to disk = N'C:\Data\Backups\AppendixC.bak'
with noformat, noinit, name = 'AppendixC - Log', compression
go

checkpoint
go

exec sys.sp_xtp_checkpoint_force_garbage_collection;
```

■ **Note** In reality, it could take more than one log backup and checkpoint event to transition CFPs to the `IN TRANSITION TO TOMBSTONE` state. You can execute the code from Listing C-10 multiple times if it happens in your system.

You can read more about the `sys.sp_xtp_checkpoint_force_garbage_collection` stored procedure at `https://msdn.microsoft.com/en-us/library/dn451428.aspx`.

Figure C-14 illustrates CFPs in the `TOMBSTONE` state. Eventually, they will be cleared from the result sets and deallocated.

	checkpoint_file_id	file_type_desc	state_desc	lower_bound_tsn	upper_bound_tsn
1	B99328BF-9FC2-47BC-86D1-708276CEDED1	DATA	ACTIVE	NULL	8
2	7B21ABEB-77E3-4775-B3B6-44CE273A713C	DELTA	ACTIVE	NULL	8
3	NULL	NULL	TOMBSTONE	NULL	NULL
4	NULL	NULL	TOMBSTONE	NULL	NULL
5	NULL	NULL	TOMBSTONE	NULL	NULL
6	NULL	NULL	TOMBSTONE	NULL	NULL

Figure C-14. *CFPs in the TOMBSTONE state*

Summary

Every checkpoint file pair transitions through various states during its lifetime. You can analyze these states using the `sys.dm_db_xtp_checkpoint_files` data management view. This view returns information about individual checkpoint files, including their type, size, state, number of inserted and deleted rows, and quite a few other properties.

The merge process merges information from the `ACTIVE` checkpoint file pairs that have a large percent of deleted rows, creating a new CFP. In most cases, you can rely on the automatic merge process; however, you can trigger a manual merge using the `sys.sp_xtp_merge_checkpoint_files` stored procedure. You can monitor the status of merge requests using the `sys.dm_db_xtp_merge_requests` view.

Merged checkpoint file pairs should be included in the log backup before they are deallocated. As with the merge, you can rely on the automatic garbage collection process in most cases. However, you can trigger the manual garbage collection process using the `sys.sp_xtp_checkpoint_force_garbage_collection` stored procedure.

APPENDIX D

In-Memory OLTP Migration Tools

This appendix discusses several SQL Server 2014 tools that help with In-Memory OLTP migration.

Management Data Warehouse Enhancements

One of the challenges during In-Memory OLTP migration is determining the list of objects that will benefit the most from it. The Pareto principle can be easily applied here: if migration targets are identified correctly, you can achieve 80 percent of possible gains by spending 20 percent of your time.

Management Data Warehouse in SQL Server 2014 has several enhancements that can help you to identify migration targets in the system. It detects the tables that suffer from lock and latch contention along with frequently executed stored procedures that consume the most CPU resources on the server. Management Data Warehouse provides a set of reports that allows you to estimate the amount of migration work and performance gain you will achieve after it is done.

Let's go through the process and configure Management Data Warehouse in the system. You can collect metrics from SQL Server 2008 and 2012 instances as long as you are using Management Data Warehouse from SQL Server 2014.

■ **Note** In this appendix, I am using the demo application and `WebRequests*_Disk` tables from Chapter 2 of this book. I also added several LOB columns and trigger to the tables to illustrate how tools provide information about constructs that are not supported in In-Memory OLTP.

You can configure Management Data Warehouse in the *Management* ➤ *Data Collection* section of SQL Server Management Studio, as shown in Figure D-1.

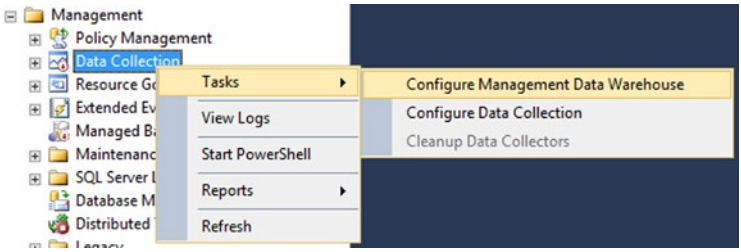


Figure D-1. *The Configure Management Data Warehouse menu*

In the first step in the process you need to choose the server and database where you will store the collected data. You can choose an existing database or create a new one, as shown in Figure D-2.

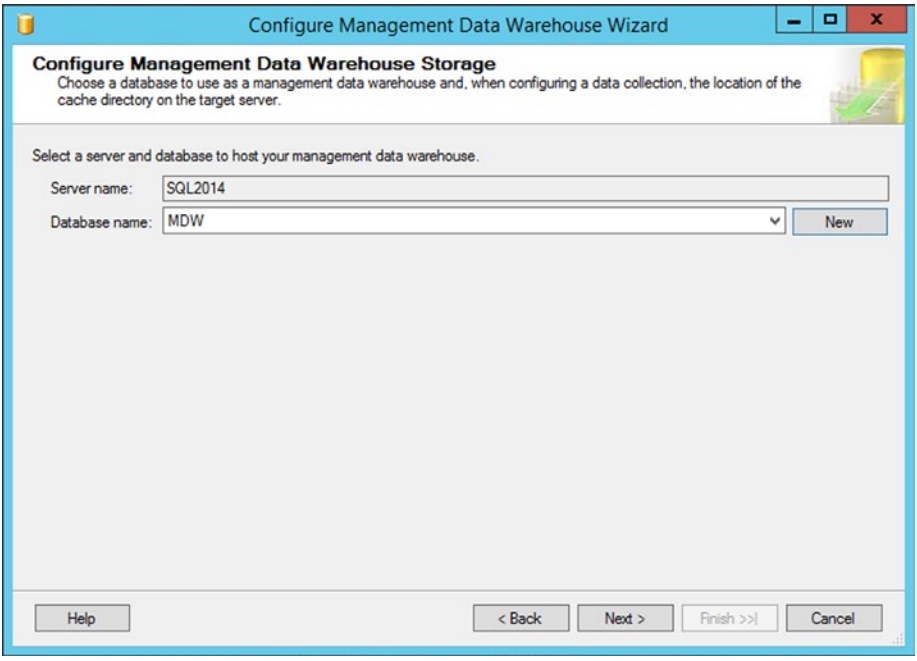


Figure D-2. *Selecting the server and database for Management Data Warehouse*

After the server and the database are selected, you can setup Management Data Warehouse security by assigning logins to the database roles, as shown in Figure D-3.

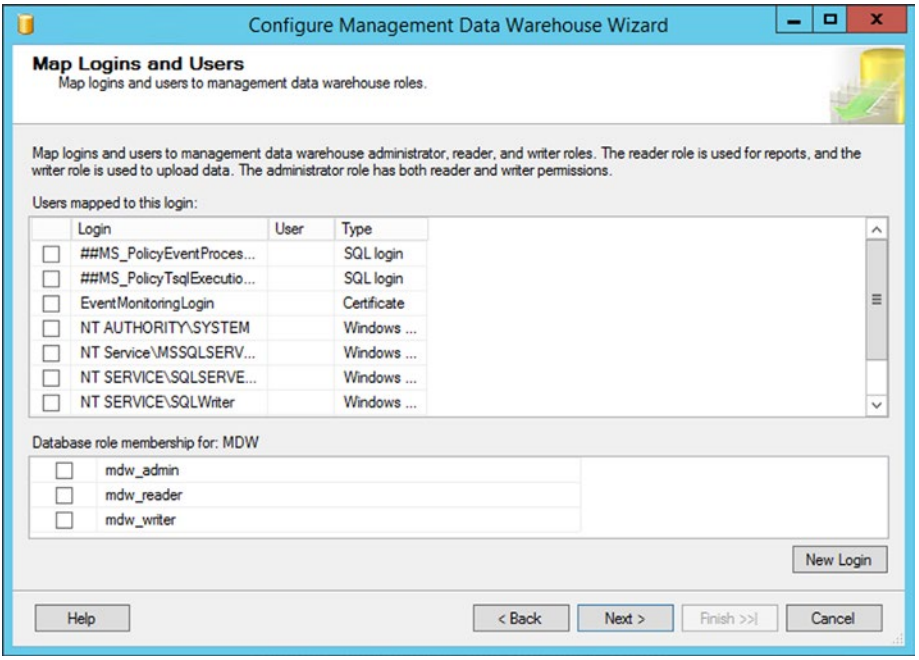


Figure D-3. *Configuring Management Data Warehouse security*

This is the final configuration step of the wizard, and clicking the *Next* button will bring you to the confirmation page. Click the *Finish* button; successful execution will bring the *Success* page shown in Figure D-4.

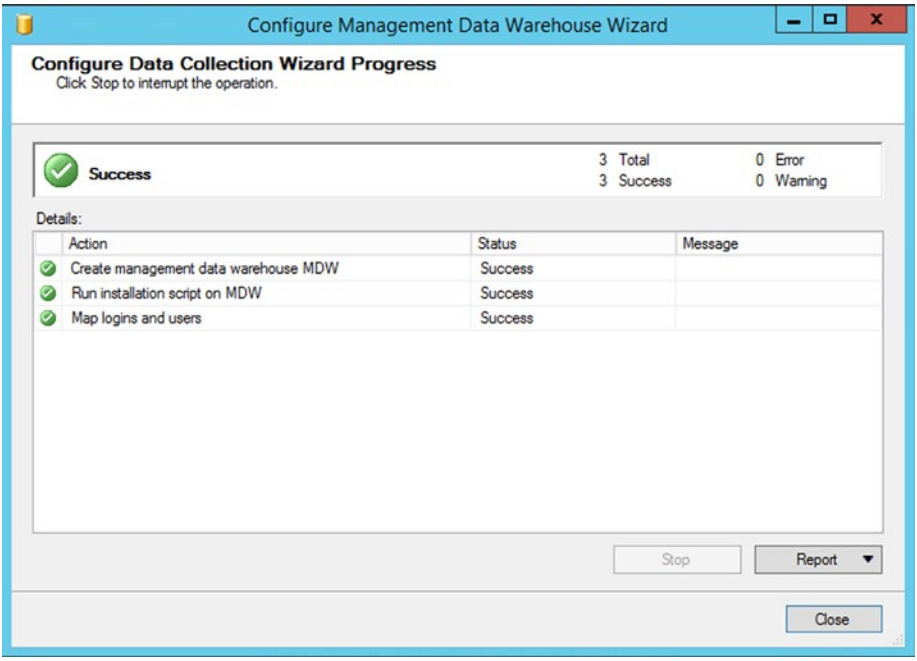


Figure D-4. *Configuring Management Data Warehouse - Success confirmation*

After Management Data Warehouse is created, you should configure and start the Data Collectors by completing another wizard from the *Management* ➤ *Data Collection* menu. Figure D-5 illustrates its location.

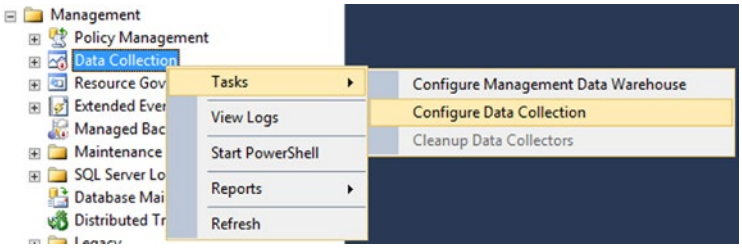


Figure D-5. *The Configure Data Collection menu*

In this wizard, you should provide connection information to Management Data Warehouse and choose *Transaction Performance Data Collection Sets* in the list of the data collectors, as shown in Figure D-6.

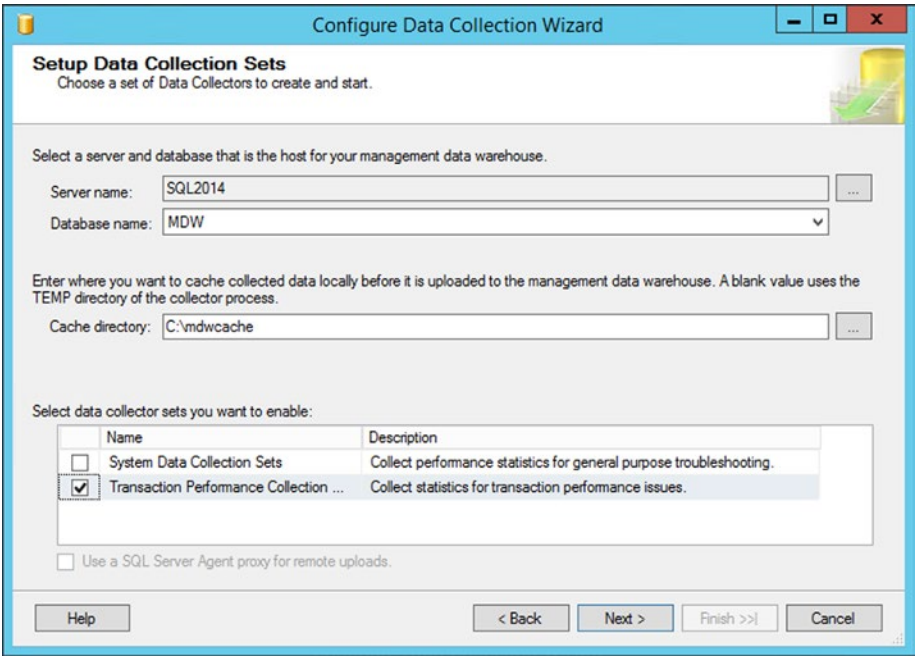


Figure D-6. *The Configure Data Collection Wizard*

After the wizard is completed, you will see two Data Collection Sets, as shown in Figure D-7. Make sure that both of them are started and collecting the information.

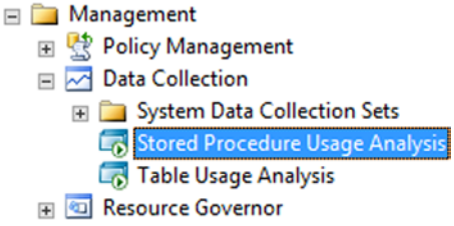


Figure D-7. *Data Collection Sets*

You can analyze collected data by using the *Transaction Performance Analysis Overview* report, which is available in the Management Data Warehouse database, as shown in Figure D-8.

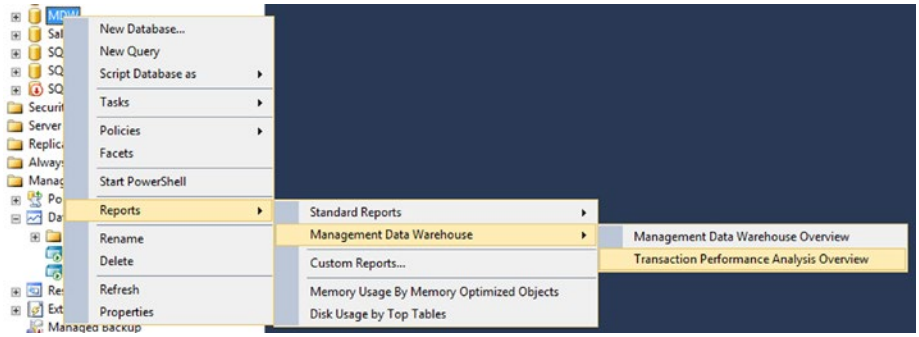


Figure D-8. *Management Data Warehouse reports*

The Transaction Performance Analysis Overview report is shown in Figure D-9.

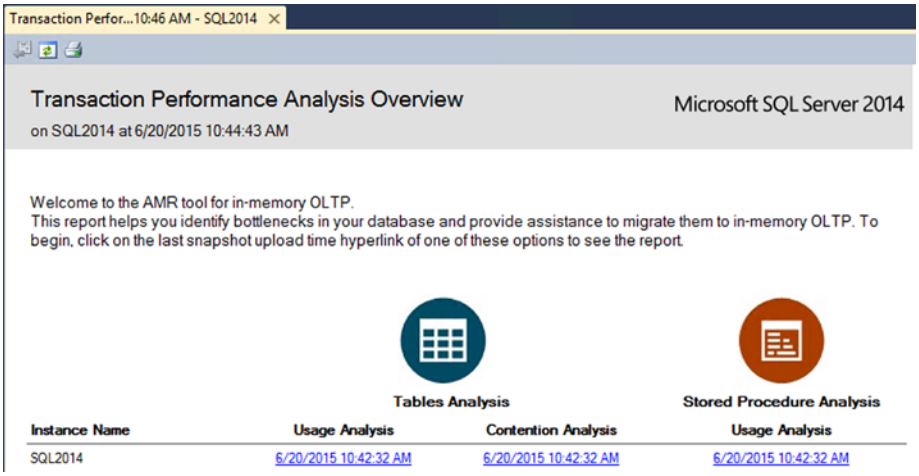


Figure D-9. *The Transaction Performance Analysis Overview report*

From this page, you have access to three drill-down reports. *Tables Usage Analysis* and *Table Contention Analysis* provide table-related statistics based on how often tables are accessed and how much they suffer from lock and latch contention.

Figure D-10 illustrates the output of the *Table Contention Analysis* report. As you can see, it displays the output in four quadrants based on the amount of work required for the migration and the estimated performance gain it will provide. Migration of the objects from the upper right quadrant will provide the most performance gain with the lowest amount of work involved.

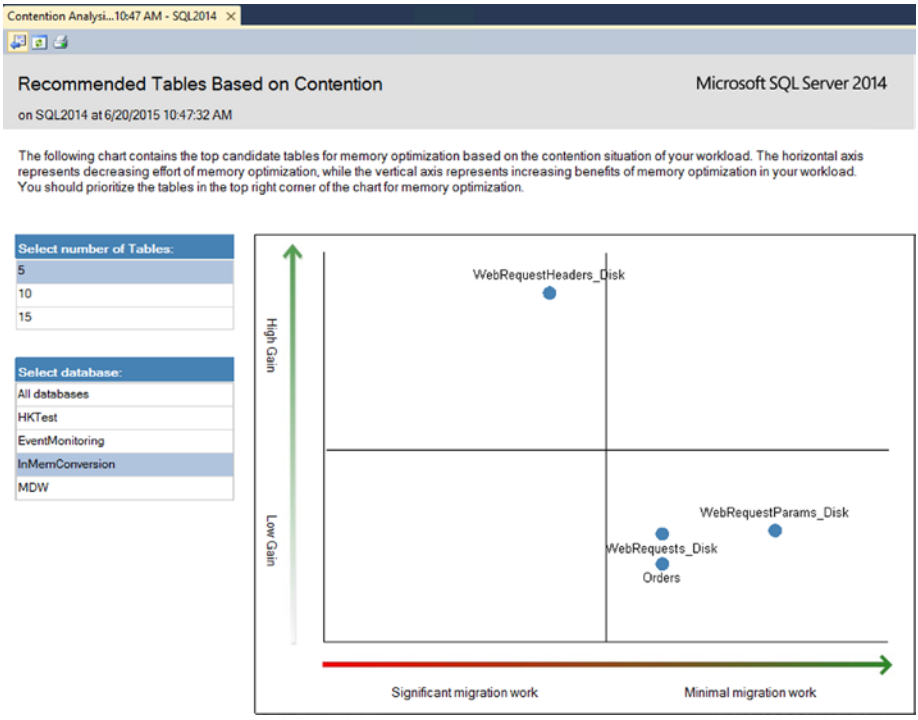


Figure D-10. *The Table Contention Analysis report*

You can see the statistics on the table level by clicking the object in the report. Figure D-11 shows the details for the `WebRequestHeaders_Disk` table in the system. The first output illustrates access method-related statistics. The demo application does not read the data from the table, which affects the numbers you see in the Figure.

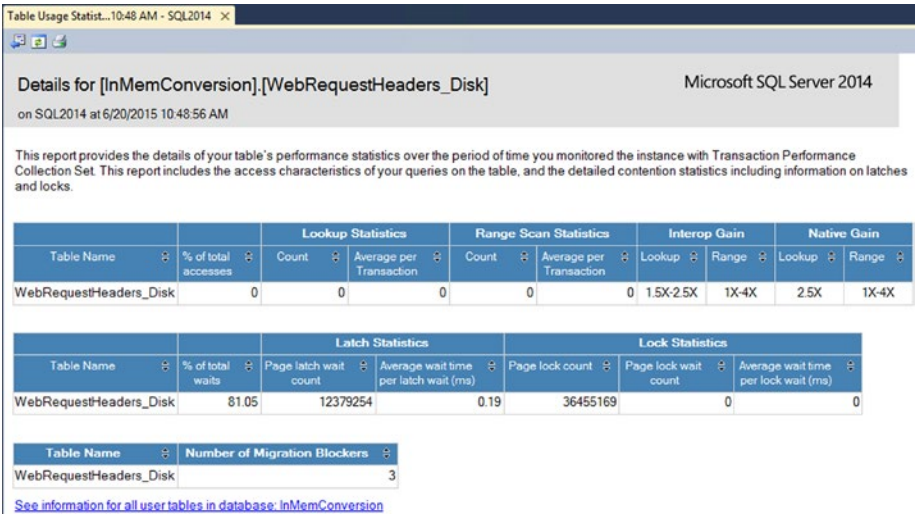


Table Usage Statist...10:48 AM - SQL2014

Details for [InMemConversion].[WebRequestHeaders_Disk]

Microsoft SQL Server 2014

on SQL2014 at 6/20/2015 10:48:56 AM

This report provides the details of your table's performance statistics over the period of time you monitored the instance with Transaction Performance Collection Set. This report includes the access characteristics of your queries on the table, and the detailed contention statistics including information on latches and locks.

		Lookup Statistics		Range Scan Statistics		Interop Gain		Native Gain	
Table Name	% of total accesses	Count	Average per Transaction	Count	Average per Transaction	Lookup	Range	Lookup	Range
WebRequestHeaders_Disk	0	0	0	0	0	1.5X-2.5X	1X-4X	2.5X	1X-4X

		Latch Statistics		Lock Statistics		
Table Name	% of total waits	Page latch wait count	Average wait time per latch wait (ms)	Page lock count	Page lock wait count	Average wait time per lock wait (ms)
WebRequestHeaders_Disk	81.05	12379254	0.19	36455169	0	0

Table Name	Number of Migration Blockers
WebRequestHeaders_Disk	3

See information for all user tables in database: InMemConversion

Figure D-11. *Table-Level Statistics*

The second output shows lock- and latch-related statistics for the table. The table suffers from a large amount of page latches, as you saw in Chapter 2.

Finally, the third output illustrates the number of migration blockers and issues that need to be addressed before migration.

Similarly, the *Procedure Usage Analysis* report shows stored procedure usage based on CPU time consumed. Figure D-12 illustrates the output of the report. The demo application called just the single procedure, which is displayed here.

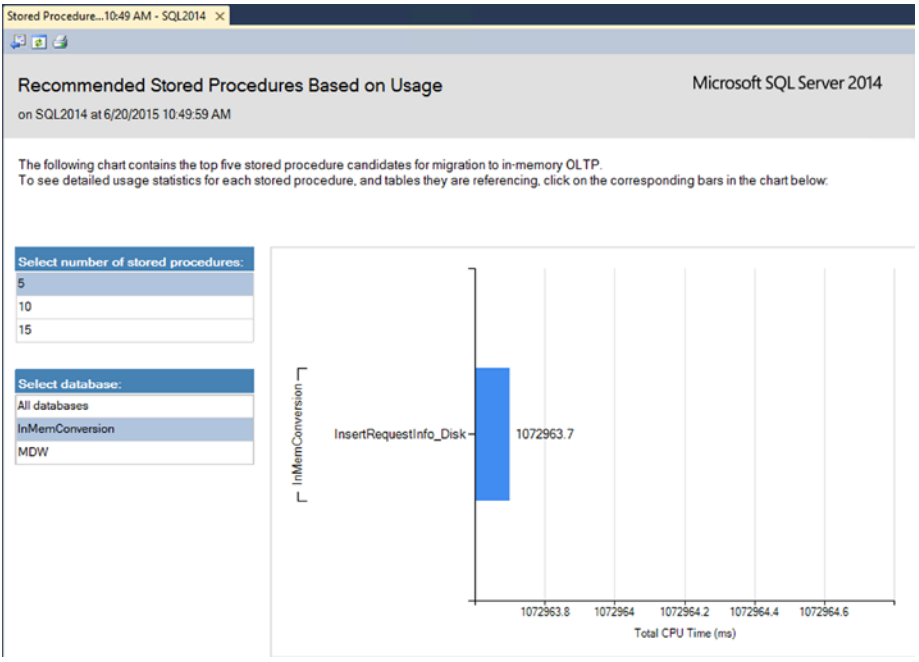


Figure D-12. *The Procedure Usage Analysis Report*

You can drill down to the procedure-level statistics, which displays the execution count, execution time metrics, and tables that are referenced by the stored procedure. Figure D-13 illustrates this page.

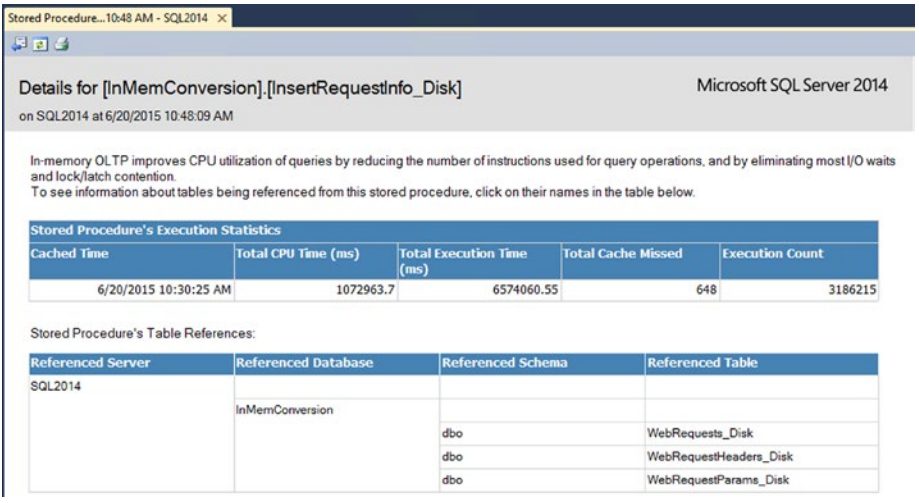


Figure D-13. *Procedure-Level Statistics*

Management Data Warehouse is a great tool that can help you identify objects that will benefit from migration. However, you should not rely solely on its results. Look and analyze the entire system before making any decisions.

Finally, it is worth mentioning that, as with any tool, the quality of output greatly depends on the quality of input. You need to collect a representative workload from a production server to get accurate results.

Memory Optimization and Native Compilation Advisors

In addition to Management Data Warehouse, SQL Server 2014 includes two other tools that can help with In-Memory OLTP migration. The *Memory Optimization* and *Native Compilation Advisors* analyze database tables and stored procedures to identify unsupported constructs. Moreover, the *Memory Optimization Advisor* can perform the actual migration, creating an In-Memory OLTP filegroup and memory-optimized table, and move data from the on-disk table there.

You can access both advisors from the object context menu in SSMS. Figure D-14 shows table context menu with the *Memory Optimization Advisor* menu item highlighted.

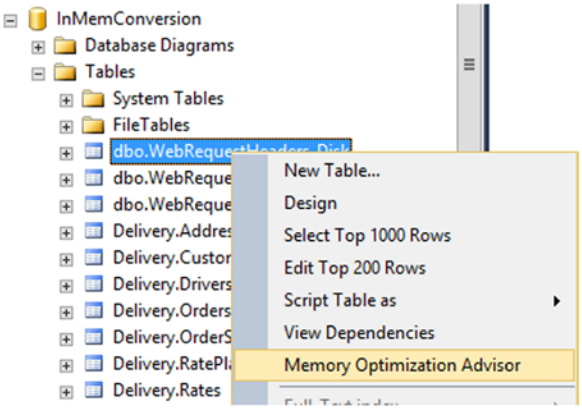


Figure D-14. *The Memory Optimization Advisor menu*

As the first step, the wizard analyzes the table and displays constructs that are unsupported by In-Memory OLTP. Figure D-15 shows the output of the validation on the `WebRequestHeaders_Disk` table. As mentioned, I added several LOB columns and a trigger to the table, which were reported by the advisor.

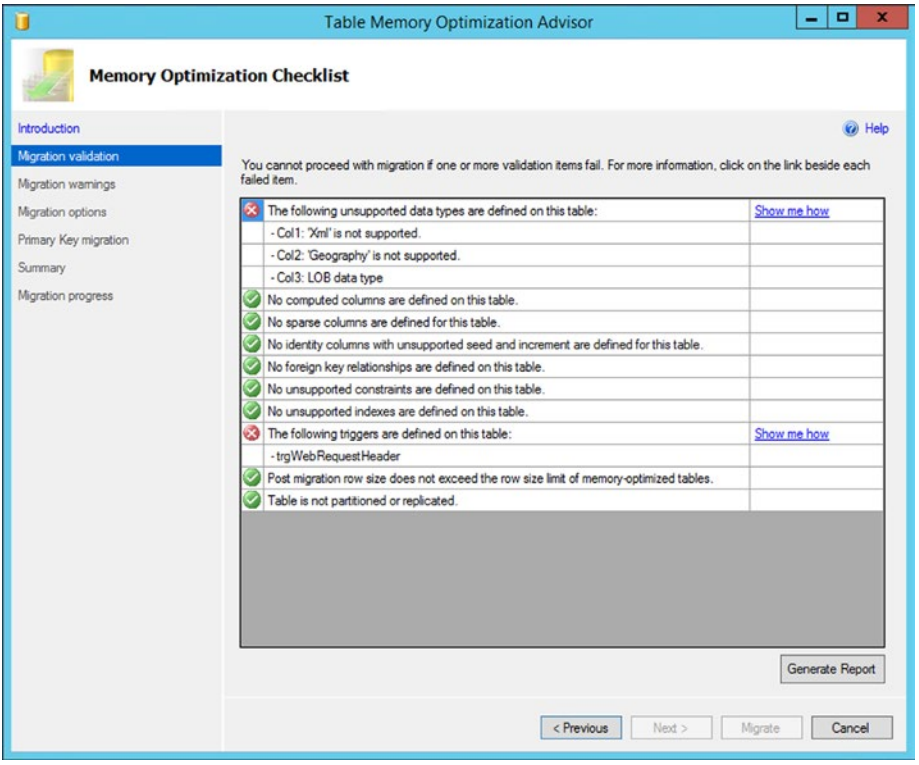


Figure D-15. *The Memory Optimization Advisor validation results*

If the table does not use any unsupported constructs, the advisor proceeds with the option of creating an In-Memory OLTP filegroup and performing actual table migration.

The simplicity of the wizard, however, is a two-edged sword. It can simplify the migration process and, in some cases, allow the enabling of In-Memory OLTP and moving data into memory with a few mouse clicks. However, as you already know, In-Memory OLTP deployments require careful hardware and infrastructure planning, redesigning of indexing strategies, changes in database maintenance and monitoring, and quite a few other steps to be successful. Improperly done migration can lead to suboptimal results, and the simplicity of the advisor increases that chance.

The advisor is a very useful tool for identifying migration roadblocks. You should be very careful, however, to rely on it performing the actual migration process.

As the opposite of the *Memory Optimization Advisor*, the *Native Compilation Advisor* does not create a natively compiled version of the stored procedures. It just analyzes whether stored procedures have unsupported constructs that prevent native compilation.

Figure D-16 illustrates the output of the *Native Compilation Advisor* for the `InsertRequestInfo_Disk` stored procedure defined in Chapter 2.

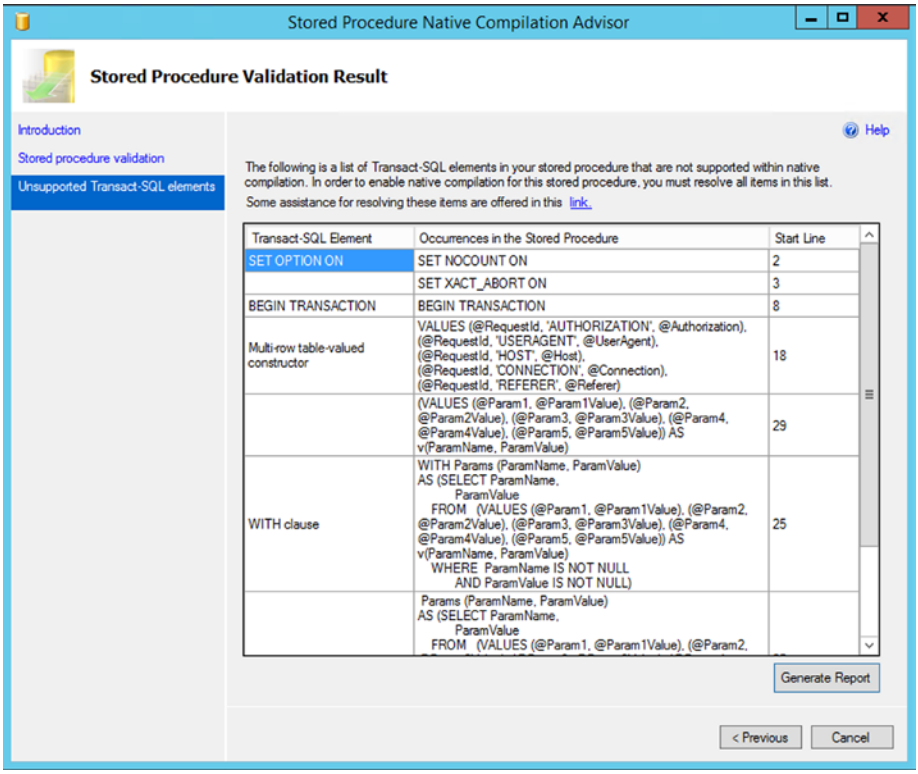


Figure D-16. *Native Compilation Advisor output*

In-Memory OLTP migration tools can help you identify targets for migration and help during the process. However, it is best to take their advice with a grain of salt and not explicitly rely on their output. After all, you know your system better than any automatic tool does.

Summary

SQL Server 2014 provides several tools that can help with In-Memory OLTP migration. Management Data Warehouse allows you to collect transaction performance metrics and identify the possible targets for migration. The *Memory Optimization* and *Native Compilation Advisors* analyze tables and stored procedures to identify the constructs unsupported by In-Memory OLTP.

Those tools are beneficial and can save you a good amount of time during the migration process. However, you should not rely strictly on their output when you perform the analysis. You need to analyze the entire system, including infrastructure and hardware, indexing strategies, database maintenance routines, and other factors to achieve the best results with In-Memory OLTP.

Index

A

Administration and monitoring tasks
  execution statistics, 159
  in-memory OLTP transactions, 157
  memory-optimized tables, 153
  Resource Governor
    classification process, 151
    internal and default resource pools, 151
    recovery process, 152
    resource pools, 151–152
Autocommitted transactions, 113

B

Buffer pool, 29

C

Checkpoint file pairs (CFPs), 121, 219
  ACTIVE checkpoint file, 227
  database creation, 222
  dbo.HKData table, 226
  forcing CHECKPOINT, 225
  in TOMBSTONE state, 231
  lifetime of, 220
  log backup and garbage collection, 231
  memory-optimized table, 222
  merge requests, 228
  on disk, 224
  sys.db_dm_xtp_checkpoint_files view, 219
  UNDER CONSTRUCTION, 223
Commit dependency, 116
Cross-container transactions
  autocommitted transactions, 113
  incompatible transaction isolation levels, 112
  SNAPSHOT isolation, 112
  transaction levels, 112

D, E, F

Data-loading process, 132
Data storage
  CFP, 121
    ACTIVE CFP state, 125
    CHECKPOINT process, 125
    INSERT and DELETE, 122
    IN TRANSITION TO TOMBSTONE state, 127
    MERGED SOURCE CFP state, 126
    MERGE TARGET state, 126
    PRECREATED CFP state, 124
    REQUIRED FOR BACKUP/HA state, 127
    TOMBSTONE state, 127
    UNDER CONSTRUCTION CFP state, 125
  multiple CFPs, 123
  on-disk tables, 121
Deployment and management
  administration and monitoring tasks, 151
  hardware components, 147
    CPU, 148
    I/O performance, 148
    memory, 149
Dusty corners scan, 139

G

Garbage collection process
BeginTs and EndTs timestamps, 135
cooperative and scalable, 135
data management views, 140
DELETE operation, 135
dusty corners scan, 139
generations, 138
idle worker thread, 136
idxLinkCount element, 136
memory-optimized table, 140
memory statistics
after scan, 143
idle worker thread cycle, 144
table creation, 142
table deletion, 143
non-blocking, 135
responsive, 135
SELECT operation, 137
summary statistics, 144
UPDATE operation, 135
worker queues, 139
workflow, 139
Generations, 138
GetRow() method, 11, 92

H

Hash indexes
bucket_count
change option, 47
creation, 42
data selection, 44
hash table lookup, 41
sys.dm_db_xtp_hash_index_stats, 43
definition, 39
SARGability
index seek operation, 50
test tables creation, 49
Hash indexes *vs.* nonclustered indexes
point lookup performance
data selection, 76
execution time, 77
tables creation, 74

I, J, K

Idle worker thread, 136
In-memory OLTP
advantages, 1
8,060-byte maximum row size limit, 171
case-insensitive collections, 182
dbo.SplitData, 174
execution plan, 174
INSTEAD OF triggers, 173
outer join, 173
Picture and Description columns, 172
Products table, 172
SplitData function, 176
unique and foreign key constraints, 176
catalog views, 162
cost/benefits analysis, 169
data management views, 162
checkpoint operations, 165
garbage collection, 165
memory usage statistics, 164
object and index statistics, 163
transaction management, 164
engine architecture, 4
Enterprise Edition feature, 171
extended events, 165
goals, 2
importing batch of rows, 185
client code, 187
memory-optimized table type, 187
Table, TVP and stored procedures, 186
limitations, 1
Management Data Warehouse, 233
memory-optimized tables
ETL performance, 191
indexing of, 170
nonclustered indexes, 194, 196
query parallelism, 191
scan performance, 190
stored procedures, execution time, 190
UPDATE performance, 195
with on-disk temporary objects, 188
mixed workloads, 203
performance counters, 167
performance improvements, 169
session/object state-store, 196
dedicated storage/cache, 197
ObjStoreDataAccess class, 201
ObjStoreService class, 202
ObjStoreUtils class, 199
replicate content, 197
scalability issues, 197
Session Store implementation, 197
SQL Server 2014 Management Studio, 170

In-Memory OLTP objects
database creation, 7
latches (*see* Latches)
memory-optimized tables
creation, 9
durability setting, 10
hash indexes, 10
limitations, 10
natively complied stored procedure, 13
nonclustered indexes, 10
performance counters, 22–23
T-SQL stored procedure, 11
wait statistics, 22–23
InterlockedCompareExchange mechanism, 209

L

Latches
features, 16
on-disk tables
creation, 17
performance counters, 21
transaction log, 24
wait statistics, 21
Log Buffer, 128

M

Management Data Warehouse, 233
data collection sets, 237
Memory Optimization and Native Compilation Advisors, 242
menu configuration, 233
procedure-Level statistics, 241
security configuration, 235
server and database, 234
success confirmation, 236
Table Contention Analysis report, 238
Table-Level statistics, 240
Transaction Performance Analysis Overview report, 237
Usage Analysis report, 240
Memory-optimized tables
availability groups, 38
BeginTs timestamp, 32–33
constraints, 37
creation, 30
data row
halloween effect, 34
index pointers, 34
payload, 34
data updation, 32
EndTs timestamp, 32–33
hash indexes, 40
limitations, 37
native complication, 35
vs. on-disk tables
clustered index, 28
nonclustered index, 29
statistics
data distribution, 55
DBCC SHOW_STATISTICS statement, 52
execution plans, 56
nested loop join algorithm, 53
table creation, 54
test queries, 55
updation, 57
supported data types, 36
variables, 99
Memory pointers management, 209
data modifications and concurrency, 210
InterlockedCompareExchange function, 209
Merge Policy Evaluator, 126
Mixed abstract tree (MAT), 80

N

Native compilation
interop mode performance comparison
data insertion, 94
delete operations, 98
select operations, 98
test tables creation, 93
update operations, 97
stored procedure (*see* T-SQL stored procedure)
T-SQL features
CAST, 88
control flow, 86
CONVERT, 88
date/time functions, 88
error functions, 88
ISNULL, 88
math functions, 88

Native compilation (*cont.*)
NEWID, 88
NEWSEQUENTIALID, 88
operators, 87
query surface area, 87
@@ROWCOUNT, 88
SCOPE_IDENTITY, 88
string functions, 88
Nonclustered indexes
Bw-Tree indexes, 67
creation, 62
definition, 61
delta record, 70
index scan operations, 63
index seek operations, 63
leaf pages, 69
page merging, 215
page splitting, 213
sorting order
execution plans, 66–67
index key column, 65–66
on-disk table creation, 64
sys.dm_db_xtp_index_stats view, 72

O

Optimistic concurrency, 105

P, Q

Page merging, 215
Page splitting, 213
Pareto principle, 233
Pessimistic concurrency, 105
Phantom Read phenomenon, 108
Pure imperative tree (PIT), 80

R, S

Recovery, 131

T, U, V

Transaction isolation levels, 104
concurrency phenomena, 104
dirty reads, 104
in-memory OLTP, 106
memory-optimized table, 106
Phantom Read phenomenon, 108
REPEATABLE READ, 106
SERIALIZABLE, 106
SNAPSHOT, 106
Non-Repeatable Reads, 104
optimistic concurrency, 105
pessimistic concurrency, 105
Phantom Reads, 104
REPEATABLE READ, 107
SERIALIZABLE, 108
shared locks, 105
SNAPSHOT, 105, 109
Transaction logging, 128
COMMIT, 128
in-memory OLTP transaction, 129
log buffer, 128
on-disk table modification, 130
UNDO and REDO, 128
Transaction processing, 103
atomicity, 103
consistency, 103
durability, 104
isolation, 104
lifetime
COMMIT request, 116
data modification operations, 115
insert operation, 115
memory-optimized tables, 114
scan set, 116
validation phase, 117
write set, 116
T-SQL stored procedure
atomic blocks
object creation, 89
write/write conflict, 90
C code, 81
creation, 84
DLL, 81
file types, 82
folder, 82
interpretation, 92
limitations, 86
MAT, 80
objects list, 83
optimization, 91
PIT, 80
query optimizer, 80

W, X, Y, Z

Write-ahead logging (WAL), 128